

University of Oxford

College of Computer Science

Elzein Salah Elzein

22/12/2021

PHD Report Draft

Written By Elzein Salah Elzein

Safe Multi-Agent Reinforcement Learning for Autonomous Robotic Teams

Contents (Section)	Page
1. Introduction	2
2. Literature Review	44
3. Distributed Control of Robotic Networks	180
4. Research Overview	349
5. Methodology	649
6. Related Work	725
7. Discussion	872
8. Conclusion	901
9. References	951

1. Introduction

Chapter 1

A Brief Review of Supervised Learning

There are a number of algorithms that are typically used for system identification, adaptive control, adaptive signal processing, and machine learning. These algorithms all have particular similarities and differences.

However, they all need to process some type of experimental data. How collect the data and process it determines the most suitable algorithm

to use. In adaptive control, there is a device referred to as the *self-tuning regulator*. In this case, the algorithm measures the states as outputs, estimates the model parameters, and outputs the control signals. In reinforcement learning, the algorithms process rewards, estimate value functions, and output actions. Although one may refer to the recursive least squares (RLS) algorithm in the self-tuning regulator as a supervised learning algorithm and reinforcement learning as an unsupervised learning algorithm, they are both very similar. In this chapter, we will present a number of well-known baseline supervised learning algorithms.

1.1 Least Squares Estimates

The least squares (LS) algorithm is a well-known and robust algorithm for fitting experimental data to a model. The first step is for the user to define a mathematical structure or model that he/she believes will fit the data. The second step is to design an experiment to collect data under suitable conditions. “Suitable conditions” usually means the operating conditions under which the system will typically operate. The next step is to run the estimation algorithm, which can take several forms, and, finally, validate the identified or “learned” model. The LS algorithm is often used to fit the data. Let us look at the case of the classical two-dimensional linear regression fit that we are all familiar with:

$$y(n) = ax(n) + b$$

1.1

In this a simple linear regression model, where the input is the sampled

signal $x(n)$ and the output is $y(n)$. The model structure defined is a straight line. Therefore, we are assuming that the data collected will fit a straight line. This can be written in the form

$$y(n) = \phi^T \theta \quad 1.2$$

where $\phi^T = [x(n) \ 1]$ and $\theta^T = [a \ b]$. How one chooses ϕ determines the model structure, and this reflects how one believes the data should

have. This is the essence of machine learning, and virtually all university students will at some point learn the basic statistics of linear regression. Behind the computations of the linear regression algorithm is the scalar cost function, given by

$$V = \sum_{n=1}^N (y(n) - \phi^T(n)\hat{\theta})^2 \quad 1.3$$

The term $\hat{\theta}$ is the estimate of the LS parameter θ . The goal is for the estimate $\hat{\theta}$ to minimize the cost function V . To find the “optimal” value of the parameter estimate $\hat{\theta}$, one takes the partial derivative of the cost function V with respect to $\hat{\theta}$ and sets this derivative to zero. Therefore, one gets

$$\begin{aligned} \frac{\partial V}{\partial \hat{\theta}} &= \sum_{n=1}^N (y(n) - \phi^T(n)\hat{\theta})\phi(n) \\ &= \sum_{n=1}^N \phi(n)y(n) - \sum_{n=1}^N \phi(n)\phi^T(n)\hat{\theta} \end{aligned} \quad 1.4$$

Setting $\frac{\partial V}{\partial \hat{\theta}} = 0$, we get

$$\sum_{n=1}^N \phi(n)\phi^T(n)\hat{\theta} = \sum_{n=1}^N \phi(n)y(n) \quad 1.5$$

Solving for $\hat{\theta}$, we get the LS solution

$$\hat{\theta} = \left[\sum_{n=1}^N \phi(n)\phi^T(n) \right]^{-1} \left[\sum_{n=1}^N \phi(n)y(n) \right] \quad 1.6$$

where the inverse, $\left[\sum_{n=1}^N \phi(n)\phi^T(n) \right]^{-1}$, exists. If the inverse does not exist, then the system is not identifiable. For example, if in the straight line case one only had a single point, then the inverse would not span the two-dimensional space and it would not exist. One needs at least two

independent points to draw a straight line. Or, for example, if one had exactly the same point over and over again, then the inverse would not exist. One needs at least two independent points to draw a straight line. The matrix $\left[\sum_{n=1}^N \phi(n)\phi^T(n) \right]$ is referred to as the *information matrix* and is related to how well one can estimate the parameters. The inverse of the information matrix is the covariance matrix, and it is proportional to the variance of the parameter estimates. Both these matrices are positive definite and symmetric. These are very important properties which are used extensively in analyzing the behavior of the algorithm. In the literature, one will often see the covariance matrix referred to as $P = \left[\sum_{n=1}^N \phi(n)\phi^T(n) \right]^{-1}$. We can write the second equation on the right of Eq. (1.4) in the form

$$\frac{\partial V}{\partial \hat{\theta}} = 0 = \sum_{n=1}^N (y(n) - \phi^T(n)\hat{\theta})\phi(n) \quad 1.$$

and one can define the prediction errors as

$$\epsilon(n) = (y(n) - \phi^T(n)\hat{\theta}) \quad 1.8$$

The term within brackets in Eq. (1.7) is known as the *prediction error* or, as some people will refer to it, the *innovations*. The term $\epsilon(n)$ represents the error in predicting the output of the system. In this case, the output term $y(n)$ is the correct answer, which is what we want to estimate. Since we know the correct answer, this is referred to as *supervised learning*.

Notice that the value of the prediction error times the data vector is equal to zero. We then say that the prediction errors are orthogonal to the data, or that the data sits in the null space of the prediction errors. In simplistic terms, this means that, if one has chosen a good model structure $\phi(n)$, then the prediction errors should appear as white noise. Always plot the prediction errors as a quick check to see how good your predictor is. If the errors appear to be correlated (i.e., not white noise), then you can improve your model and get a better prediction.

One does not typically write the linear regression in the form of Eq. (1.2), it typically will add a white noise term, and then the linear regression takes the form

$$y(n) = \phi^T(n)\theta + v(n) \quad 1.9$$

where $v(n)$ is a white noise term. Equation (1.9) can represent an infinite number of possible model structures. For example, let us assume that we want to learn the dynamics of a second-order linear system or the parameters of a second-order infinite impulse response (IIR) filter. Then we could choose the second-order model structure given by

$$y(n) = -a_1y(n-1) - a_2y(n-2) + b_1u(n-1) + b_2u(n-2) + v(n) \quad 1.10$$

Then the model structure would be defined in $\phi(n)$ as

$$\phi^T(n) = [y(n-1) \quad y(n-2) \quad u(n-1) \quad u(n-2)] \quad 1.11$$

In general, one can write an arbitrary k th-order autoregressive exogenous (ARX) model structure as

$$y(n) = -a_1y(n-1) - a_2y(n-2) - \dots - a_my(n-k) + b_1u(n-1) + b_2u(n-2) + \dots + b_{n-k}u(n-k) + v(n) \quad 1.12$$

and $\phi(n)$ takes the form

$$\phi^T(n) = [y(n-1) \quad \dots \quad y(n-m) \quad u(n-1) \quad \dots \quad u(n-m)] \quad 1.13$$

One then collects the data from a suitable experiment (easier said than done!), and then computes the parameters using Eq. (1.6). The vector $\phi(n)$ can take many different forms; in fact, it can contain nonlinear functions of the data, for example, logarithmic terms or square terms, and it can have different delay terms. To a large degree, one can use one's professional judgment as to what to put into $\phi(n)$. One will often write the data in the matrix form, in which case the matrix is defined as

$$\Phi = [\phi(1) \quad \phi(2) \quad \dots \quad \phi(N)] \quad 1.14$$

and the output matrix as

$$Y = [y(1) \quad y(2) \quad \dots \quad y(N)] \quad 1.15$$

Then one can write the LS estimate as

$$\hat{\Theta} = (\Phi\Phi^T)^{-1} \Phi Y \quad 1.16$$

Furthermore, one can write the prediction errors as

$$E = Y - \Phi^T \hat{\Theta} \quad 1.1$$

We can also write the orthogonality condition as $\Phi E = 0$.

The LS method of parameter identification or machine learning is very well developed and there are many properties associated with the technique. In fact, much of the work in statistical inference is derived from the few equations described in this section. This is the beginning of many scientific investigations including work in the social sciences.

1.2 Recursive Least Squares

The LS algorithm has been extended to the RLS algorithm. In this case, the parameter estimate is developed as the machine collects the data in real time. In the previous section, all the data was collected first, and then the parameter estimates were computed on the basis of Eq. (1.6). The RLS algorithm is derived by assuming a solution to the LS algorithm and then adding a single data point. The derivation is shown in Reference [1]. In the RLS implementation, the cost function takes a slightly different form. The cost function in this case is

$$V = \sum_{n=1}^N \lambda^{(N-n)} (y(n) - \phi^T(n) \hat{\theta})^2 \quad 1.18$$

where $\lambda \leq 1$. The term λ is known as the *forgetting factor*. This term will place less weight on older data points. As such, the resulting RLS algorithm will be able track changes to the parameters. Once again, taking the partial derivative of V with respect to $\hat{\theta}$ and setting the derivative to zero, we get

$$\hat{\theta} = \left[\sum_{n=1}^N \lambda^{(N-n)} \phi(n) \phi^T(n) \right]^{-1} \left[\sum_{n=1}^N \lambda^{(N-n)} \phi(n) y(n) \right] \quad 1.19$$

The forgetting factor should be set as $0.95 \leq \lambda \leq 1.0$. If one sets the forgetting factor near 0.95, then old data is forgotten very quickly; the rule of thumb is that the estimate of the parameters $\hat{\theta}$ is approximately based on $1/(1 - \lambda)$ data points.

The RLS algorithm is as follows:

$$\begin{aligned}\hat{\theta}(n+1) &= \hat{\theta}(n) + L(n+1)(y(n+1) - \phi^T(n+1)\hat{\theta}(n)) \\ L(n+1) &= \frac{P(n)\phi(n+1)}{\lambda + \phi^T(n+1)P(n)\phi(n+1)} \\ P(n+1) &= \frac{1}{\lambda} \left(P(n) - \frac{P(n)\phi(n+1)\phi^T(n+1)P(n)}{\lambda + \phi^T(n+1)P(n)\phi(n+1)} \right)\end{aligned}\tag{1.20}$$

One implements Eq. (1.20) by initializing the parameter estimator $\hat{\theta}$ to the user's best initial estimate of the parameters, which is often simply zero. The covariance matrix P is typically initialized to a relatively large diagonal matrix, and represents the initial uncertainty in the parameter estimate.

One can implement the RLS algorithm as in Eq. (1.20), but the user should be careful that the covariance matrix P is always positive definite and symmetric. If the P matrix, because of numerical error by repeatedly computing the RLS, ceases to be positive definite and symmetric, then the algorithm will diverge. There are a number of well-developed algorithms to ensure that the P matrix remains positive definite. One can use a square-roots approach whereby the P matrix is factored into its Cholesky factorization or the UDU factorization. Such methods are described in Reference [1].

Let us examine Eq. (1.20) and notice that the update to the parameter estimate is the previous estimate plus a matrix $L(n)$ times the current prediction error. We will see this structure in almost every algorithm that will be described in machine learning. In this case, we have an actual correct answer, which is the measurement $y(n)$, and we call such algorithms *supervised learning*.

1.3 Least Mean Squares

In the field of signal processing, there are a few commonly used techniques to model or characterize the dynamics of a communications channel and then compensate for the effects of the channel on the signal. These techniques are referred to as *channel equalization* and *echo cancellation*. There are numerous books on adaptive signal processing and adaptive filtering [2]. Most of these techniques use the least mean

squares (LMS) approach to identify the coefficients of a model of the channel. Once again, as in the LS and RLS algorithms, we must choose an appropriate model structure to define the communication channel dynamics. In the field of signal processing, one would typically use what is known as a *finite impulse response* (FIR) filter as the underlying model structure that describes the system. To maintain consistency with the previous section, one can write the channel dynamics as

$$y(n) = b_0 u(n) + b_1 u(n-1) + \cdots + b_k u(n-k) + v(n) \quad 1.21$$

where $y(n)$ is the output of the filter, or the communications channel, at time step n , b_i are the filter coefficients that we want to estimate or *learn*, and $u(n)$ is the input signal. Typically, the signal $u(n)$ is the communication signal that we want to recover from the output signal $y(n)$. We define an error signal

$$e(n) = y(n) - \hat{y}(n) \quad 1.22$$

where $\hat{y}(n) = \phi^T(n)\hat{\theta}$. This is the same signal as the prediction error in Eq. (1.8). The LMS algorithm defines the cost function as the expected value of the prediction errors as

$$J(n) = E[\epsilon^2(n)] \quad 1.23$$

We can write the squared error term as

$$\begin{aligned} \epsilon^2(n) &= (y(n) - \phi^T(n)\hat{\theta})^2 \\ &= y^2(n) - 2y(n)\phi^T(n)\hat{\theta} + \hat{\theta}^T \phi(n)\phi^T(n)\hat{\theta} \end{aligned} \quad 1.24$$

We take the expectation and get

$$E[\epsilon^2(n)] = E[y^2(n)] - 2\hat{\theta}^T E[y(n)\phi(n)] + \hat{\theta}^T E[\phi(n)\phi^T(n)]\hat{\theta} \quad 1.25$$

We then define the variance $\sigma_y^2 = E[y^2]$ as the mean squared power, and the cross-correlation vector is defined as $p = E[y(n)\phi(n)]$. Then we define the information matrix, which is almost the same matrix as in Section 1.1, as $R = E[\phi(n)\phi^T(n)]$. If the system statistics are stationary, that is, the statistics do not change, the terms σ_y^2 , p , and R , are constants and the cost function, as a function of changing $\hat{\theta}$, will have the shape of a bowl. The cost function $J(n)$ can be written as

$$J(n) = \sigma_y^2 - 2\hat{\theta}^T p + \hat{\theta}^T R \hat{\theta} \quad 1.26$$

Once again, as in Eq. (1.4), to find the *optimal* parameter estimate $\hat{\theta}$ to minimize the cost function, we take the partial derivative of the cost function $J(n)$ with respect to $\hat{\theta}$, and determine the value of $\hat{\theta}$ that sets the partial derivative to zero. We can take the partial derivative of $J(n)$ as

$$\frac{\partial J(n)}{\partial \hat{\theta}} = \frac{\partial \sigma_y^2}{\partial \hat{\theta}} - 2 \frac{\partial \hat{\theta}^T p}{\partial \hat{\theta}} + \frac{\partial \hat{\theta}^T R \hat{\theta}}{\partial \hat{\theta}} \quad 1.2$$

We then compute the partial derivative for each of the terms on the right-hand side of Eq. (1.27). Taking each term separately, we get

$$\begin{aligned} \frac{\partial \sigma_y^2}{\partial \hat{\theta}} &= 0 \\ 2 \frac{\partial \hat{\theta}^T p}{\partial \hat{\theta}} &= 2p \\ \frac{\partial \hat{\theta}^T R \hat{\theta}}{\partial \hat{\theta}} &= 2R\hat{\theta} \end{aligned} \quad 1.28$$

Substituting into Eq. (1.27), we get

$$\frac{\partial J(n)}{\partial \hat{\theta}} = -2p + 2R\hat{\theta} = 0 \quad 1.29$$

Solving for $\hat{\theta}$, we get the solution for the optimal parameter estimate as

$$\theta^* = R^{-1}p \quad 1.30$$

Equation (1.30) is the well-known Wiener solution. However, the Wiener solution in Eq. (1.30) requires the computation of the inverse of a large matrix R . Notice the similarity between the Wiener solution and the LS solution in Eq. (1.6). Let us say that we want to estimate the expectations in Eq. (1.25); then we would get the average by computing

$$R_{avg} = \left[\frac{1}{N} \sum_{n=1}^N \phi(n) \phi^T(n) \right]$$

$$p_{avg} = \left[\frac{1}{N} \sum_{n=1}^N \phi(n) y(n) \right]$$

wh

Substituting the above values into Eq. (1.30), we get the LS solution given Eq. (1.6). In essence, the LMS Wiener solution and the LS solution are essentially the same.

ve
In the world of signal processing and in particular adaptive signal processing, the processing speed is very important. Furthermore, the model structure used in adaptive signal processing, especially for communication applications, can have many parameters. It would not be unusual to have 200 terms in the $\phi(n)$ vector, which means the term k in Eq. (1.21) would be $k = 200$. In that case, the R matrix will be 200×200 , which would be prohibitively large to take the inverse of in Eq. (1.30). As such, a gradient *steepest descent* method is normally implemented. This is a very common technique throughout the fields of engineering and is very similar to the well-known Newton–Raphson method of finding the zeros and roots of various functions. The steepest descent method is an iterative method. The idea is to start with an initial guess of the parameter values: often one will simply choose zero for the parameter values. In the lexicon of signal processing, one would refer to the parameters as *tap weights*. Then one iteratively adjusts the parameters such that one moves down the cost function along the gradient. Let us say that the current estimate of the parameter vector is $\hat{\phi}(now)$; then we compute the next value of the parameter vector as

$$\hat{\theta}(next) = \hat{\theta}(now) - \mu g \quad 1.32$$

where g is the gradient and is given by the derivative of the cost function with respect to the parameter estimation vector, $\hat{\theta}$, as defined in Eq. (1.29). Then, substituting for g in Eq. (1.32), we get

$$\hat{\theta}(next) = \hat{\theta}(now) - \mu 2p - \mu 2R\hat{\theta}(now) \quad 1.33$$

In recursive form, it is written as

$$\hat{\theta}(n+1) = \hat{\theta}(n) - \mu 2p - \mu 2R\hat{\theta}(n) \quad 1.34$$

We can also write Eq. (1.34) in the form

$$\hat{\theta}(n+1) = (I - \alpha R)\hat{\theta}(n) - \alpha p \quad 1.35$$

where $\alpha = 2\mu$. One may recognize, from systems theory, that if the eigenvalues of $(I - \alpha R)$ are less than 1, then the recursion in Eq. (1.35)

will converge. This places a limit on the step size of the steepest descent method. We will come back to this point in the next section when we look at the stochastic approximation methods. The effect of step size is an important parameter in machine learning algorithms.

The difficulty in computing the recursion in Eq. (1.34) is the computation of the statistical terms R and p , where R is the information matrix or the autocorrelation matrix, and p is the cross-correlation matrix. Their statistics are often unknown and have to be estimated as we did in Eq. (1.31). However, these estimates are computationally intensive and one has to wait until N data points are collected. Instead, the LMS algorithm proposes that one estimate these matrices based on a single data point at each sampling time, as

$$\begin{aligned} \hat{R}(n) &= \phi(n)\phi^T(n) \\ \hat{p}(n) &= \phi(n)y(n) \end{aligned} \quad 1.36$$

This is sometimes referred to as the *dirty gradient* method or the *stochastic gradient* method. The idea is that one has to descend in the general direction of the gradient and not exactly along the gradient. Think of yourself walking down a hill; you can either go straight down or, if it is really steep, you may choose to go back and forth traversing the hill, much the same way as a skier. Either way, you end up at the bottom of the hill. Now we substitute the estimates for $\hat{R}$ and $\hat{p}$ given by Eq. (1.36) into the recursive equation given by Eq. (1.34) and we get

$$\hat{\theta}(n+1) = \hat{\theta}(n) + 2\mu\phi(n)y(n) - 2\mu\phi(n)\phi^T(n)\hat{\theta}(n) \quad 1.3$$

Now we factor out the term $2\mu\phi(n)$ and get the standard LMS recursive algorithm as

$$\hat{\theta}(n+1) = \hat{\theta}(n) + 2\mu\phi(n)(y(n) - \phi^T(n)\hat{\theta}(n)) \quad 1.38$$

Recall that the term in brackets on the right-hand side, given by

$(y(n) - \phi^T(n)\hat{\theta}(n))$, is the prediction error or the innovation. The term $\phi(n)\hat{\theta}(n)$ is the current prediction of the output $y(n)$. If we compare the RLS algorithm in Eq. (1.20) to Eq. (1.38), we see that the update has a similar form. The update to the parameters is the previous estimate plus a matrix vector times the prediction error. In fact, it can be shown that at the stationary point, or the value at which the covariance matrix update takes the value $P(n+1) = P(n)$ in Eq. (1.20), the LMS algorithm is equivalent to the RLS algorithm for a particular set of parameters.

There is a vast literature on various implementations and convergence results associated with the LMS algorithm, but the key element for this book is that the machine learns the parameters of a preconceived model of the system based on the available experimental data and knowledge of the correct answer given by $y(n)$. The new parameter is the old parameter plus a vector based on the data times the known error in the predicted output.

1.4 Stochastic Approximation

The method of stochastic approximation is an older method of system identification. In fact, it is a method of finding the zeros of a function and is very similar to both the RLS and LMS methods, and it is the fundamental structure for the Q-learning algorithms associated with reinforcement learning and much of the machine learning literature. The early work in stochastic approximation comes from the work by Robbins and Monro [3] and Wolfowitz [4]. A good textbook on the topic is by Kushner and Yin [5]. Monro formulated the problem as finding the level at which a continuous function $M(\theta) = \alpha$. Writing the problem in the form $M(\theta) - \alpha = 0$ converts it into the problem of finding the zeros of a function. If one knows the gradient of the function, then one can use the well-known Newton–Raphson method to find the zeros, but in this case one takes the noise-corrupted measurements of the function at different values of θ . One then makes small corrections to θ in the estimated direction of zero.

The method of stochastic approximation and the theoretical proofs of stability are used in the proofs of convergence for several fundamental algorithms in reinforcement learning. Formulating the problem in a similar form to the previous sections, we get the function

$M(\theta) = (y(\theta) - \phi^T \theta) = 0$ and we can write the prediction error and the error in getting to zero as $\epsilon = (y - \phi^T \theta)$. The stochastic approximation algorithm is

Id

$$\theta(n+1) = \theta(n) - a_n(y - \phi^T \theta) \quad 1.39$$

where a_n is a variable step size that goes to zero such that

$$0 < \sum_1^{\infty} a_n^2 = A < \infty \quad 1.40$$

An

References

- [1] L. Ljung and T. Soderstrom, *Theory and Practice of Recursive Identification*. Cambridge, Massachusetts: The MIT Press, 1983.
- [2] B. Farhang-Boroujeny, *Adaptive Filters: Theory and Applications*. New York: John Wiley & Sons, Inc., 1998.
- [3] H. Robbins and S. Monroe, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. **22**, no. 3, pp. 400–407, 1951.
- [4] J. Wolfowitz, "On the stochastic approximation method of robbins and monro," *Annals of Mathematical Statistics*, vol. **23**, no. 3, pp. 457–461, 1952.
- [5] H. J. Kushner and G. G. Yin, *Recursive Approximation and Recursive Algorithms and Applications*. New York: Springer-Verlag, 2nd ed., 2003.

Chapter 2

Single-Agent Reinforcement Learning

The objective of this chapter is to introduce the reader to reinforcement learning. A good introductory book on the topic is Reference [1] and we

ll follow their notation. The goal of reinforcement learning is to maximize a reward. The interesting aspect of reinforcement learning, as ll as unsupervised learning methods, is the choice of rewards. In this chapter, we will discuss some of the fundamental ideas in reinforcement learning which we will refer to in the rest of the book. We will start with the simple n -armed bandit problem and then present ideas on the meaning of the “value” function.

2.1 Introduction

Reinforcement learning is learning to map situations to actions so as to maximize a numerical reward [1]. Without knowing which actions to take, the learner must discover which actions yield the most reward by trying them. Actions may affect not only the immediate reward but also the next situation and all subsequent rewards [1]. Different from supervised learning, which is learning from examples provided by a knowledgeable external supervisor, reinforcement learning is used for learning from interaction [1]. Since it is often impractical to obtain examples of desired behavior that are both correct and representative of all the situations, the learner must be able to learn from its own experience [1]. Therefore, the reinforcement learning problem is a problem of learning from interaction to achieve a goal.

The learner is called the *agent* or the *player* and the outside which the agent interacts with is called the *environment*. The agent chooses actions to maximize the rewards presented by the environment. Suppose we have a sequence of discrete time steps $t = 0, 1, 2, 3, \dots$. At each time step t , the agent receives a state s_t from the environment. We define a_t as the action the agent takes at time t . At the next time step, as a consequence of its action a_t , the agent receives a numerical reward $r_{t+1} \in \mathfrak{R}$ and moves to a new state s_{t+1} , as shown in [Fig. 2-1](#). At each time step, the agent implements a mapping from states to probabilities of selecting each

possible action [1]. This mapping is called the *agent's policy* and is denoted as π_t , and $\pi_t(s, a)$ is the probability of $a_t = a$ at $s_t = s$.

Reinforcement learning methods specify how the agent changes its policy as a result of its experience to maximize the total amount of reward it receives over the long run [1].

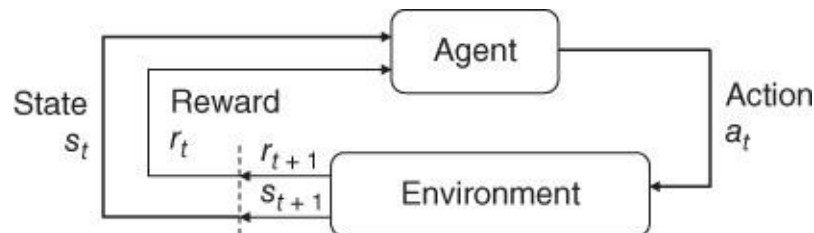


Figure 2-1 Agent–environment interaction in reinforcement learning.

Reproduced from [1], with permission of MIT Press.

reinforcement learning problem can be studied under the framework of stochastic games [2]. Such a framework contains two simpler frameworks: Markov decision processes (MDPs), and matrix games [2]. MDPs involve a single agent and multiple states, while matrix games include multiple agents and a single state. Combining MDPs and matrix games, stochastic games are considered as reinforcement learning problems with multiple agents and multiple states.

2.2 n -Armed Bandit Problem

The n -armed bandit problem is taken from playing slot machines. The idea is that one is playing a slot machine with n arms. Each arm will give you a different reward, or a different probability of winning or losing. The idea is to determine which of the n arms will give you the greatest reward. Therefore, there are n actions that one can take, each of the actions representing one of the arms that you can pull. The action that gives the greatest reward is referred to as the *greedy* reward. How would one learn which is the best arm to pull to get the greatest expected reward? You would probably try and pull each arm many times and try to compute a running average of which arm seemed to give you the most reward. We will define the value of each arm as the expected reward for the arm, or action. In this case, pulling a given arm is the same as choosing an action. Let us estimate the value of an action at trial t as

$$Q_i(a) = \frac{r_1 + r_2 + \dots + r_k}{k} \quad 2.1$$

where r_i is the reward of choosing arm a on the i th time step. However, there were t actions played in total, but only k of those times was the arm or action a played. We will denote the actual value of the arm or action a as $Q^*(a)$. The rule for choosing which action to take is to always choose the *greedy* action except for the relatively small probability of ϵ of choosing a random action. The *greedy* actions are taken based on which action has the highest expected reward at that particular timestep. However, sometimes one should explore to find out if some other choice of action would be better. This random selection of some action that is not the *greedy* one is referred to as *exploration*. Within machine learning, there is a kind of tension between how much *exploration* there should be and how much *exploitation* there should be.

Let us take a simple example of the 10-armed bandit. We start by assigning a random reward for each of the 10 arms from a normal random distribution with mean zero and variance 1. We take 10 numbers from the distribution $N(0, 1)$. These 10 numbers will represent the true value or the expected reward for each of the 10 arms. We get the true reward for each arm as

$$Q^*(a) = [-0.4 \quad 1.3 \quad 0.04 \quad 0.53 \quad -0.15 \quad -1.01 \quad 0.2 \quad 1.48 \quad 0.36 \quad -0.5] \quad 2.2$$

We know that the best choice of action is action 8, with $Q^*(8) = 1.48$. This is the correct answer that we want the machine to learn. We also set the *exploration* variable to $\epsilon = 0.2$. This means that there is a 20% chance that on any given choice the machine will randomly choose an action without regard to the estimate of the reward.

We start the learning process by guessing at the expected reward for each arm by choosing our initial guess from the same normal distribution that chose the true rewards from, and the true rewards are given in Eq. (2.2). We then get our first estimate of the true reward as

$$Q_{est}(0) = [0.05 \quad 0.86 \quad -0.96 \quad 0.73 \quad 1.98 \quad -1.19 \quad -0.66 \quad 0.82 \quad 1.97 \quad -0.13] \quad 2.3$$

Based on the initial estimate, we should choose action 5, which is estimated by Eq. (2.3) as $Q_{est}(5) = 1.98$, but the true value of action 5 is really $Q^*(5) = -0.15$, and therefore the initial *greedy* choice we took is a very poor choice. The machine chooses action 5, and the resulting reward

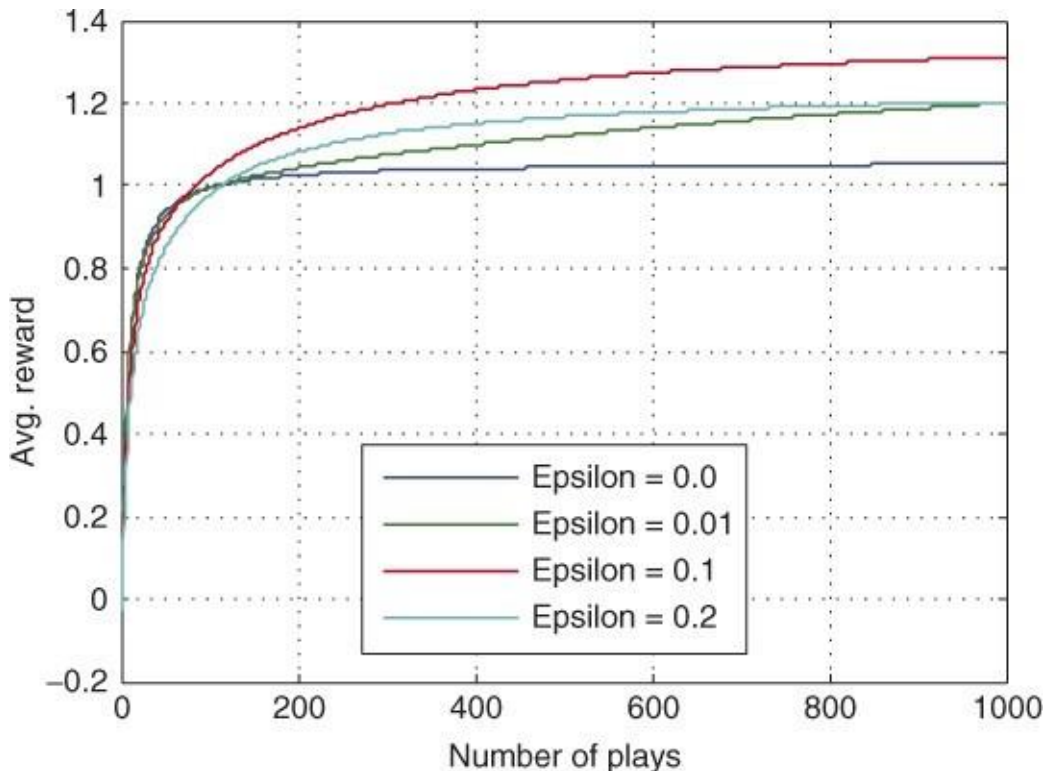
is selected from a random distribution of $N(-0.15, 1)$, which has a mean of -0.15 and a variance of 1. The machine gets the reward $Q_1(5) = -0.76$. The machine then updates the $Q_{est}(a)$ table or vector according to Eq. (2.1) as

$$Q_{est}(1) = [0.05 \quad 0.86 \quad -0.96 \quad 0.73 \quad -0.76 \quad -1.19 \quad -0.66 \quad 0.82 \quad 1.97 \quad -0.13] \quad 2.4$$

The machine then chooses another arm based on its most recent estimate of the rewards given by Eq. (2.4). The machine once again makes the *greedy* choice and chooses action 9 because $Q_{est}(9) = 1.97$. The machine chooses action $a = 9$, and the machine gets the reward $Q_2(9) = 1.50$. We then get the estimate of $Q_{est}(2)$ as

$$Q_{est}(2) = [0.05 \quad 0.86 \quad -0.96 \quad 0.73 \quad -0.76 \quad -1.19 \quad -0.66 \quad 0.82 \quad 1.5 \quad -0.13] \quad 2.5$$

We repeat this process and, on trial number 6, the machine explores and arbitrarily chooses action 6, which we know is not a good choice, because from Eq. (2.2) the true average reward for action $a = 6$ is $Q^*(6) = -1.01$. The reward the machine gets for choosing action $a = 6$ on the sixth trial is $r = 1.44$. The learning process continues until the 18th trial, when it finally recognizes action $a = 8$ as the best choice. The results of implementing the 10-armed bandit problem for different values of ϵ is illustrated in Fig. 2-2



2.3 The Learning Structure

In this section, we will formulate the basic structure for reinforcement learning. We will take the case of a single agent. The agent, or robot, is in a particular state and in that state it can take one of a number of actions. In some cases, there exists a particular action that is in some sense optimal for that particular state. We will sometimes refer to that action as the *greedy* action and the set of actions the agent should *optimally* take in each state is the action *policy* or the *strategy*. The terms *policy* and *strategy* are sometimes used interchangeably and sometimes they may have slightly different meanings: it depends on the context. If in a particular state the agent should choose a particular action, we will refer to that as a *pure strategy*, whereas if the agent should choose an action with a particular probability, then we refer to that as a *distributed policy*, or a mixed strategy. For example, if you are playing the rock-paper-scissors game, then the optimal strategy is to take each action with a probability of $1/3$, but if you knew in advance that your opponent will always choose Rock, then your *optimal* strategy or policy is to choose the pure strategy of paper.

We will define the actions as $a_t \in A(s_t)$. The action is a member of an action set for the given state. The possible actions an agent can take depend on the state the agent is in. Furthermore, there is a defined number of possible states. Here we have $s_t \in S$. If it is currently time t , our goal will be to maximize future rewards, given by

$$R_t = r_{t+1} + r_{t+2} + \dots + r_T \quad .6$$

where T is the terminal time. This type of reward function works well when we have episodic games that have a well-defined terminal time. However, in many cases we do not have a terminal time and the game continues indefinitely, such as a regulator in control applications. In those cases, without well-defined terminal times, one typically uses discounted future rewards. In this case, the reward is defined as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad 2.$$

ere γ is the discount factor given as $0 \leq \gamma \leq 1$. If γ is close to zero, then refer to the algorithm as *myopic*, but if γ is close to 1, then the algorithm is maximizing future rewards.

In reinforcement learning, we want the current state to be a good basis for predicting the future. Such systems are called *MDPs*. A decision system is said to have the Markov property when the current state is all one needs to know to make decisions about what to do next and into the future. What to do in the next step does not depend on what happened in the past, but just the state the system is currently in. For example, a game of checkers depends only on the current state of the game and not how one got to that position. A more engineering example is that the future flight of a ball depends only on its current position and velocity vector; it does not depend on how it got there. In the Markov world, we can define the Markov property as the probability of receiving a particular reward, and changing to another state is completely defined by the current state and action as $Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\}$; whereas in the non-Markovian

world the same probability would be written as $Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0\}$. We can define the probability of taking action a and transitioning from state s to the next state s' as $P_{ss'}^a = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$. Furthermore, we can define the expected reward as $R_{ss'}^a = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$.

2.4 The Value Function

The value function, in the framework of reinforcement learning defines *how good* a particular state is. The measure of how good a given state is, is based on the expected future rewards that will be achieved from that state. The players in the game follow a strategy $\pi(s, a)$, which is the probability of taking action a when in state s . The value of a state s is the expected future return of following policy $\pi(s, a)$ starting from s , and can be written as

$$e \quad V^\pi(s) = E_\pi\{R_t | s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \quad 2.8$$

We also define the action value function for policy or strategy, π :

$$Q^\pi(s, a) = E_\pi \{ R_t | s_t = s, a_t = a \} \quad 2.9$$

$$= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}$$

The term $Q^\pi(s, a)$ is a little different from $V^\pi(s)$. The term $Q^\pi(s, a)$ is the expected return if we choose action a in state s and then follow policy or strategy π afterwards. We can write the value function as

$$V^\pi(s) = E_\pi \{ R_t | s_t = s \} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\} \quad 2.10$$

$$= E_\pi \left\{ r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\} \quad 2.11$$

$$= E_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s \right\}. \quad 2.12$$

We can then write this in the form

$$\begin{aligned} V^\pi(s) &= \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a \left(R_{ss'}^a + \gamma E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s \right\} \right) \\ &= \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V^\pi(s')). \end{aligned} \quad 2.13$$

The first sum on the right-hand side of Eq. (2.13) represents the sum over all the possible actions in state s , where $\pi(s, a)$ is the probability of taking a particular action. The second sum in Eq. (2.13) is taken over all the possible next states. Recall that $P_{ss'}^a$ is the probability of transitioning from state s to state s' given one has chosen action a . The last term in brackets on the right represents the immediate reward $R_{ss'}^a$ plus the future discounted rewards from the next state onwards, given by $V^\pi(s')$.

Equation (2.13) is known as the *Bellman equation* for $V^\pi(s)$. Similarly, we can write the state-action value function as

$$Q(s, a) = \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V^\pi(s')) \quad 2.14$$

2.5 The Optimal Value Functions

define the optimal state-value function as

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in S \quad 2.15$$

and, similarly, the optimal state-action value function is given as

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \quad 2.16$$

As such, we are searching for the policy, or the choice of actions that will give the greatest reward. Then $Q(s, a)$ can be written as

$$\text{WR} \quad Q(s, a) = E \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a \right\} \quad 2.1$$

Then the optimal value becomes

$$Q^*(s, a) = E \{ r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a \} \quad 2.18$$

The term $Q^*(s, a)$ represents one getting an immediate reward of r_{t+1} for taking action a and then following the optimal policy thereafter. We can write the value function in the following form:

$$V^*(s) = \max_{a \in A(s)} Q^*(s, a) \quad \underline{2.19}$$

The value function and the state-action function look to the future. They refer to the expected future rewards. Therefore, if we know the maximum future rewards from the next state onwards, then all we have to do is to choose the best action right now. We can write Eq. (2.19) as

$$V^*(s) = \max_{a \in A(s)} Q^{\pi^*}(s) \quad 2.20$$

$$= \max_{a \in A(s)} E_{\pi^*} \{ R_t | s_t = s, a_t = a \}$$

$$= \max_{a \in A(s)} E_{\pi^*} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}$$

$$= \max_{a \in A(s)} E_{\pi^*} \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a \right\}$$

$$= \max_{a \in A(s)} E_{\pi^*} \{ r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a \}$$

We

$$= \max_{a \in A(s)} \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V^*(s_{t+1})).$$

Similarly we can write

$$Q^*(s, a) = E \left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a \right\} \quad 2.21$$

2.5.1 The Grid World Example

will take an example of a 3×3 grid, as shown in Fig. 2-3. The robot or agent can be in one of the nine cells at any starting time. It can then move in one of four directions. The actions then are taken from the set $a \in [up, down, right, left]$. When the robot or learning agent moves to cell 1, it then immediately moves to cell 9 and gets a reward of $r = +10$. If the agent hits a wall, it remains in its current cell and gets a negative reward (punishment) of $r = -1$. We will use Eq. (2.20) to derive the nine equations and nine unknowns. We will solve Eq. (2.20) with the following parameters: the discount factor for future rewards $\gamma = 0.9$, and the action selection strategy is $\pi(s, a) = 0.25$; in other words, the agent will arbitrarily choose to go either up, down, left, or right with equal probability. There are four possible actions to transition from a particular state. The only reward is when the agent gets to state 1 and then transitions with 100% probability to state 9 and receives a reward of +10. The rewards are all zero except for $r_{19}^{up} = 10, r_{19}^{down} = 10, r_{19}^{left} = 10, r_{19}^{right} = 10$ and, when the agent hits the wall, $r_{22}^{up} = -1, r_{33}^{up} = -1, r_{33}^{right} = -1, r_{44}^{left} = -1, r_{66}^{right} = -1, r_{77}^{down} = -1, r_{77}^{left} = -1, r_{99}^{down} = -1, r_{99}^{right} = -1$

$r_{22}^{up} = -1, r_{33}^{up} = -1, r_{33}^{right} = -1, r_{44}^{left} = -1, r_{66}^{right} = -1, r_{77}^{down} = -1, r_{77}^{left} = -1, r_{99}^{down} = -1, r_{99}^{right} = -1$. The transition probabilities are as follows.

$$\begin{aligned}
 P_{19}^{up} &= 1, & P_{19}^{down} &= 1, & P_{19}^{left} &= 1, & P_{19}^{right} &= 1 \\
 P_{22}^{up} &= 1, & P_{25}^{down} &= 1, & P_{21}^{left} &= 1, & P_{23}^{right} &= 1 \\
 P_{33}^{up} &= 1, & P_{36}^{down} &= 1, & P_{32}^{left} &= 1, & P_{33}^{right} &= 1 \\
 P_{41}^{up} &= 1, & P_{47}^{down} &= 1, & P_{44}^{left} &= 1, & P_{45}^{right} &= 1 \\
 P_{52}^{up} &= 1, & P_{58}^{down} &= 1, & P_{54}^{left} &= 1, & P_{56}^{right} &= 1 \\
 P_{63}^{up} &= 1, & P_{69}^{down} &= 1, & P_{65}^{left} &= 1, & P_{66}^{right} &= 1 \\
 P_{74}^{up} &= 1, & P_{77}^{down} &= 1, & P_{77}^{left} &= 1, & P_{78}^{right} &= 1 \\
 P_{85}^{up} &= 1, & P_{88}^{down} &= 1, & P_{87}^{left} &= 1, & P_{89}^{right} &= 1 \\
 P_{96}^{up} &= 1, & P_{99}^{down} &= 1, & P_{98}^{left} &= 1, & P_{99}^{right} &= 1.
 \end{aligned} \tag{2.22}$$

We can now substitute these into Eq. (2.13) to get the nine equations and nine unknowns for each state. Let us specifically write out the first equation for the first state, recalling that $\pi(s, a) = 0.25$, representing equal probability for the taking of each action and $\gamma = 0.9$. Writing out the $V^\pi(1)$ equation is a special case because, regardless of which action we take, the always move to state 9 and receive a reward of $r_{19}^a = +10$. Therefore, get

$$V^\pi(1) = \sum_{a=1}^4 (0.25) \left(\sum_9 P_{19}^a ((r_{19}^a + \gamma V^\pi(9))) \right) \tag{2.23}$$

The term

$$\sum_9 P_{19}^a (r_{19}^a + \gamma V^\pi(9)) = r_{19}^a + 0.9 V^\pi(9) \tag{2.24}$$

dropping the superscript on V^π for ease of writing, gives the first equation as

$$V(1) = 10 + 0.9 V(9) \tag{2.25}$$

Now we will go through the same process for state 2. In this case, we do not automatically jump to another state but we can move to one of four other states. Recall that hitting a wall will give us a negative reward of -1 . Then we can write

$$\begin{aligned}
 V(2) &= \sum_{a=1}^4 (0.25) \left(\sum_{s'} P_{2s'}^a (r_{2s'}^a + \gamma V(s')) \right) \\
 &= 0.25 \left(\sum_{s'} P_{2s'}^{up} (r_{2s'}^{up} + \gamma V(s')) \right) + 0.25 \left(\sum_{s'} P_{2s'}^{down} (r_{2s'}^{down} + \gamma V(s')) \right) \\
 &\quad + 0.25 \left(\sum_{s'} P_{2s'}^{left} (r_{2s'}^{left} + \gamma V(s')) \right) + 0.25 \left(\sum_{s'} P_{2s'}^{right} (r_{2s'}^{right} + \gamma V(s')) \right).
 \end{aligned} \tag{2.26}$$

If we examine the probabilities on the right of Eq. (2.26), we note that they take a value of either 1 or 0. For example, if we look at the term $P_{2s'}^{up}$, it only takes a value for $P_{22}^{up} = 1$, and all other state transitions for $P^{up} = 0$. Furthermore, if the agent tries to go up, it hits the wall and gets a reward of -1 . We can now rewrite equation 2.26 as

$$\begin{aligned}
 V(2) &= 0.25(-1 + 0.9V(2)) + 0.25(0 + 0.9V(5)) + 0.25(0 + 0.9V(1)) \\
 &\quad + 0.25(0 + 0.9V(3)) \\
 &= 0.225V(1) + 0.225V(2) + 0.225V(3) + 0.225V(5) - 0.25.
 \end{aligned}$$

Just for some further practice, we will write out the equation for $V(3)$. In this case, the agent will hit the wall if it moves up or right, and will receive a reward of -1 . Then we get

$$\begin{aligned}
 V(3) &= 0.25(-1 + 0.9V(3)) + 0.25(0 + 0.9V(6)) + 0.25(-1 + 0.9V(3)) \\
 &\quad + 0.25(0 + 0.9V(2)) \\
 \text{wh} \quad &= 0.225V(2) + 0.45V(3) + 0.225V(6) - 0.5.
 \end{aligned}$$

We continue going through the process of determining the equations for each state, and we then write out the equations in matrix form as $AV = B$, where A is given as

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.9 \\ -0.225 & 0.775 & -0.225 & 0 & -0.225 & 0 & 0 & 0 & 0 \\ 0 & -0.225 & 0.55 & 0 & 0 & -0.225 & 0 & 0 & 0 \\ -0.225 & 0 & 0 & 0.775 & -0.225 & 0 & -0.225 & 0 & 0 \\ 0 & -0.225 & 0 & -0.225 & 1 & -0.225 & 0 & -0.225 & 0 \\ 0 & 0 & -0.225 & 0 & -0.225 & 0.775 & 0 & 0 & -0.225 \\ 0 & 0 & 0 & -0.225 & 0 & 0 & 0.55 & -0.225 & 0 \\ 0 & 0 & 0 & 0 & -0.225 & 0 & -0.225 & 0.775 & -0.225 \\ 0 & 0 & 0 & 0 & 0 & -0.225 & 0 & -0.225 & 0.55 \end{bmatrix}$$

and B is given by

$$B^T = [10 \quad -0.25 \quad -0.5 \quad -0.25 \quad 0 \quad -0.25 \quad -0.5 \quad -0.25 \quad -0.5]$$

Solving for this set of nine equations and nine unknowns, we get the true value of each state as

$$V^T = [8.85 \quad 2.5 \quad -0.07 \quad 2.5 \quad 0.92 \quad -0.44 \quad -0.07 \quad -0.44 \quad -1.27] \quad 2.27$$

Recall that the value function for each state is for the future discounted rewards. Therefore, state 1 does not get a value of 10 but a value of 8.85 cause we do not know where the agent goes from state 9 onwards. We show the value of each state in the grid in [Fig. 2-4](#).

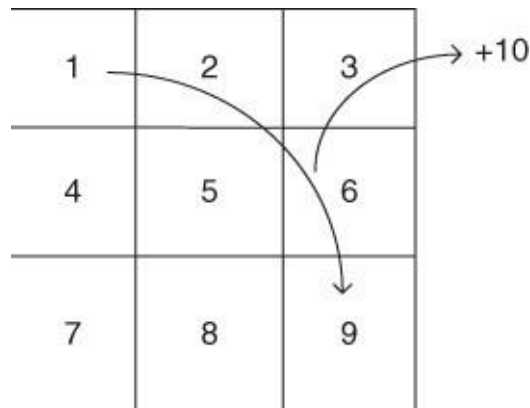


Figure 2-3 Example of the grid world.

8.85	2.50	-0.07	→ +10
2.50	0.92	-0.44	
-0.07	-0.44	-1.27	

Figure 2-4 Values for each of the states.

This is an important result as we delve into machine learning algorithms. In some way, we need to learn the value of the state. Furthermore, even for this relatively simple stochastic game, look at the amount of information we needed to know to solve for the value of the state. We needed to know all the transition probabilities from state to state, based on what action was taken, and in this case it was relatively easy; the transition probabilities were either 1 or 0. We have to know all possible actions in each state, and we need to know the rewards a priori.

The goal of machine learning is to try to automatically learn what would be the best path to take, given any initial state, such that you collect the greatest reward. You and I can look at this game and almost immediately recognize the best path to take, but what if you knew nothing about the game. How would you learn what the best path would be? You would try out some steps until you eventually hit state 1 and collected your big reward. This is what we want the agent to eventually learn how to do.

2.6 Markov Decision Processes

MDP [3] is a tuple (S, A, T, γ, R) , where S is the state space, A is the action space, $T : S \times A \times S \rightarrow [0, 1]$ is the transition function, $\gamma \in [0, 1]$ is the discount factor, and $R : S \times A \times S \rightarrow \mathbb{R}$ is the reward function. The transition function denotes a probability distribution over next states given the current state and action such that

$$\sum_{s' \in S} T(s, a, s') = 1 \quad \forall s \in S, \quad \forall a \in A \quad 2.28$$

where s' represents a possible state at the next time step. The reward

h

function denotes the received reward at the next state given the current action and the current state. An MDP has the following Markov property: the player's next state and reward depend only on the player's current state and action. A player's policy $\pi : S \rightarrow A$ is defined as a probability distribution over the player's actions from a given state. A player's policy $\pi(s, a)$ satisfies

$$\sum_{a \in A} \pi(s, a) = 1 \quad \forall s \in S \quad 2.29$$

For any MDP, there exists a deterministic optimal policy for the player, where $\pi^*(s, a) \in \{0, 1\}$ [4]. The goal of a player in an MDP is to maximize the expected long-term reward. In order to evaluate a player's policy, we have the following concept of the state-value function: The value of a state s (or the state-value function) under a policy π is defined as the expected return when the player starts at state s and follows a policy π thereafter. Then the state-value function $V^\pi(s)$ becomes

$$V^\pi(s) = E_\pi \left\{ \sum_{k=0}^T \gamma^k r_{k+t+1} | s_k = s \right\} \quad 2.30$$

where T is a final time step, t is the current time step, r_{k+t+1} is the received immediate reward at the time step $k + t + 1$, and $\gamma \in [0, 1]$ is a discount factor. In (2.30), we have $T \rightarrow \infty$ if the task is an infinite-horizon task such that the task will run over an infinite period. If the task is episodic, T is defined as the terminal time when each episode is terminated at the time step T . Then we call the state where each episode ends as the *terminal state* s_T . In a terminal state, the state-value function is always zero such that $V(s_T) = 0 \quad \forall s_T \in S$. An optimal policy π^* will maximize the player's discounted future reward for all states such that

$$V^*(s) \geq V^\pi(s) \quad \forall \pi, \forall s \in S \quad 2.31$$

The state-value function under a policy in (2.30) can be rewritten as a recursive equation called the *Bellman equation* [5] as follows:

$$V^\pi(s) = \sum_{a \in A} \pi(s, a) \sum_{s' \in S} T(s, a, s') (R(s, a, s') + \gamma V^\pi(s')) \quad 2.32$$

where $T(s, a, s') = Pr\{s_{k+1} = s' | s_k = s, a_k = a\}$ is the probability of the next state being $s_{k+1} = s'$ given the current state $s_k = s$ and action $a_k = a$ at time

step k , and $R(s, a, s') = E\{r_{k+1} | s_k = s, a_k = a, s_{k+1} = s'\}$ is the expected immediate reward received at state s' given the current state s and action a . If the player starts at state s and follows the optimal policy π^* thereafter, we have the optimal state-value function denoted by $V^*(s)$. The optimal state-value function $V^*(s)$ is also called the *Bellman optimality equation*, where

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s') (R(s, a, s') + \gamma V^*(s')) \quad 2.33$$

We can also define the action-value function as the expected return of choosing a particular action a at state s and following a policy π thereafter. The action-value function $Q^\pi(s, a)$ is given as

$$Q^\pi(s, a) = \sum_{s' \in S} T(s, a, s') (R(s, a, s') + \gamma V^\pi(s')) \quad 2.34$$

If the player chooses action a at state s and follows the optimal policy π^* thereafter, the action-value function becomes the optimal action-value function $Q^*(s, a)$, where

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s') (R(s, a, s') + \gamma V^*(s')) \quad 2.35$$

In a terminal state s_T , the action-value function is always zero such that $Q(s_T, a) = 0 \forall s_T \in S$.

2.7 Learning Value Functions

In the previous section, we computed the value function for each state by solving nine equations and nine unknowns. Also we saw in the previous section that just writing out the equations can be tedious even for the relatively simple example that we had. In this section, we will present the iterative computer algorithm that will solve the equation for the value functions. This type of approach is known more widely as *dynamic programming*. Given that we already know the transition probabilities, the action strategies, and the rewards, we can compute the value of the states as in Eq. (2.13). Recall that we wrote the equation for the value function in the form $AV = B$, and we can also write it as $AV - B = 0$, which is solved for as $V = A^{-1}B$, and most of us learned to solve this

equation using Gaussian elimination. Let us say we just write an algorithm to find the zero of $AV - B = 0$. Any dirty type gradient algorithm can work. Most people who have done technical work have done this type of thing. Sometimes they tried to find the argument of a function by trial and error. The idea is to solve Eq. (2.36) iteratively. We

ll not write out the set of equations, but we will write a machine learning algorithm to find it. Furthermore, we will construct this learning algorithm as a gradient search algorithm to find the zero of a function. In more formal terms, we are

Algorithm 2.1 Value iteration algorithm

```

Initialize  $V(s) = 0$  for all  $s \in S$ 
repeat
  Set  $\Delta = 0$ 
  For each  $s \in S$ :
     $v \leftarrow V(s)$ 
     $V(s) \leftarrow \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V_k(s'))$ 
     $\Delta = \max(\Delta, |v - V(s)|)$ 
until  $\Delta < \theta$  for all  $s \in S$  ( $\theta$  is a small positive number)

```

searching for the stationary point of a recursive algorithm, which is similar to stochastic approximation.

The algorithm works as follows: we initialize the value function to zero as $V(s) = 0 \quad \forall s$. We then compute Eq. (2.13) and repeating it for convenience as

$$V_{k+1}(s) = \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V_k(s')) \quad 2.36$$

This gives us the first guess at the value of each state. Recall, we must know $\pi(s, a)$, $P_{ss'}^a$ and $R_{ss'}^a$ in advance. We then compute the difference $\Delta = V_{k+1} - V_k$. Once this difference becomes small enough, we stop the recursion given by Eq. (2.36). Once we get to $V_{k+1} - V_k \Rightarrow 0$, then the algorithm given by the recursion in (2.36) has reached a stationary point. The value iteration algorithm is given in Algorithm 2.1. We set $\Delta = 0.001$, and the algorithm executes 40 iterations before it exits with the result

$$V^T = [8.85 \quad 2.5 \quad -0.07 \quad 2.5 \quad 0.92 \quad -0.44 \quad -0.07 \quad -0.44 \quad -1.27] \quad 2.37$$

o

h

which is the same result that we got with the exact solution in Eq. (2.27).

2.8 Policy Iteration

In the previous section, we presented a recursive algorithm to solve the n equations and n unknown problem to compute the value of each state. However, we used the known state transition matrices $P_{ss'}^a$, the known rewards $R_{ss'}^a$, and the known action strategies $\pi(s, a)$ to do the computations. However, can we determine whether a better strategy would give us a higher reward. If we searched over all the possible strategies, could we find a strategy that would yield a higher reward and what would the resulting value of the states be under that new strategy? In this section, we present the *policy iteration* algorithm. This algorithm will compute both the value of the state and the *optimal* action strategy. Recall, we wrote out the state-action value function in Eq. (2.14) as

$$Q(s, a) = \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V(s')) \quad 2.38$$

The goal we is to search for the policy that maximizes $Q(s, a)$, as

$$Q^*(s, a) = \max_a \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V(s')) \quad 2.39$$

The algorithm has two phases. In the first phase, the value of each state is computed as was done in the value iteration algorithm. In the next phase, compute Eq. (2.39) and search for the maximizing action. If more than one action yields the equivalent maximizing result, then we assign the probability of each action equivalently. For example, if two actions give the same maximizing result, then each action gets a probability of 50%. We implement the *policy iteration* Algorithm 2.2 on the previous grid game. This time we get

Algorithm 2.2 Policy iteration algorithm

```
1: Initialize  $V(s) = 0$  for all  $s \in S$ 

2: Policy Evaluation
3: repeat
4:    $\Delta = 0$ 
5:   For each  $s \in S$ :
6:      $v \leftarrow V(s)$ 
7:      $V(s) \leftarrow \sum_a \pi(s, a) \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V_k(s'))$ 
8:      $\Delta = \max(\Delta, |v - V(s)|)$ 
9:   until  $\Delta < \theta$  for all  $s \in S$  ( $\theta$  is a small positive number)

10: Policy Improvement
11: repeat
12:   For each  $s \in S$ :
13:      $b \leftarrow \pi(s)$ 
14:      $\pi(s) \leftarrow \max_a \sum_{s'} P_{ss'}^a (R_{ss'}^a + \gamma V_k(s'))$ 
15:   If  $b \neq \pi(s)$  Then go back to policy evaluation
16: until  $b = \pi(s)$ 
```

When a different strategy and the value function increases substantially. [Figure 2-5](#) illustrates the resulting strategies for each state and the associated probability. For example, state 1 has four directions and equal probability of choosing any action because the next state is always state 9 regardless of the chosen action; whereas in state 2 the optimal action is to always choose with 100% probability to take action *left* and move to state 1.

Then in state 5, the *up* and *left* actions each have equal probability of 50% ([Fig. 2-6](#)). Based on this new strategy, the value of each state can then be computed as

$$V^T = [24.4 \quad 22.0 \quad 19.8 \quad 22.0 \quad 19.8 \quad 17.8 \quad 19.8 \quad 17.8 \quad 16.0]$$

Notice the dramatic improvement in the value of each state in comparison to the values in ([2.37](#)). This is because we have now computed the value of the state based on the *optimal* action strategy.

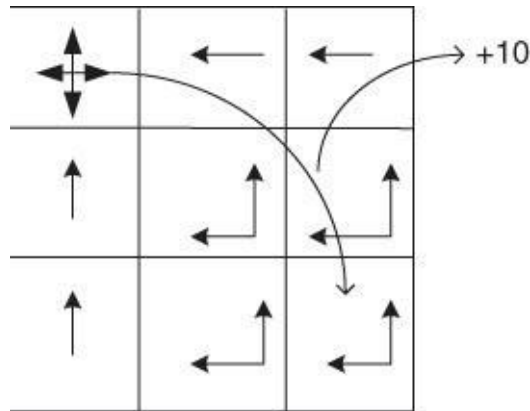


Figure 2-5 Resulting optimal policies.

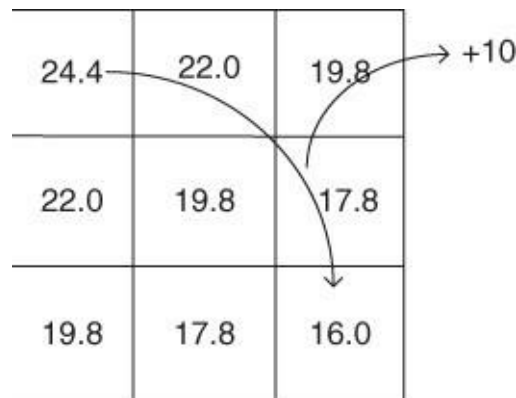


Figure 2-6 Resulting state values based on the optimal policies.

2.9 Temporal Difference Learning

The idea of reinforcement learning is to learn the best policy and actions. In Section 2.7 we iteratively learnt, or more accurately we calculated, the value of the state on the basis of the knowledge of the action strategies $\pi(s, a)$, the state transition probabilities $P_{ss'}^a$, and the rewards $R_{ss'}^a$. In Section 2.8, we presented the policy iteration algorithm that searched for the *optimal* policy and we saw how the optimal policy would improve the value of the states. However, in this case we had to know the state transition probabilities $P_{ss'}^a$ and the rewards $R_{ss'}^a$. In this section, we will present a machine learning algorithm that will learn the value of the state based on just observing the rewards from the environment.

Let us return to the original definition of the value of the state as the expected future rewards as we wrote in (2.13):

e

$$V^\pi(s) = E_\pi \{ R_t | s_t = s \} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\} \quad 2.40$$

$$= E_\pi \left\{ r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\} \quad 2.41$$

we

$$= E_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s \right\}. \quad 2.42$$

We can then write this in the form

$$V^\pi(s) = r_{t+1} + \gamma V^\pi(s') \quad 2.43$$

Notice that we have identified the value of the state with the action strategy π . If we know the transition probabilities and the rewards, then we can solve for the value of the state by either deriving the n equations and n unknowns as we did in Section 2.5.1 or by dynamic programming as we did in Section 2.7. However, in this case we do not have the luxury of knowing the transition probabilities $P_{ss'}^a$ nor the rewards $R_{ss'}^a$.

Therefore, we will use another search method that we are familiar with from the least mean square (LMS) approach. Let us begin with an initial estimate of the value function for each state. We will often, in practice, initialize the value function to zero. Let us rewrite Eq. (2.43) in the form

$$\epsilon = r_{t+1} + \gamma V(s') - V(s) \quad 2.44$$

Now we will update our estimate of the value of the state as

$$V_{k+1}(s) = V_k(s) + \alpha [r_{t+1} + \gamma V(s') - V(s)] \quad 2.45$$

This is the same structure that we saw in the case of the LMS algorithm defined in Eq. (1.38). The update is the previous value plus learning rate α (or step size) times the prediction error. The algorithm takes the following form: We apply

Algorithm 2.3 Temporal difference algorithm

```
Initialize  $V(s) = 0$  for all  $s \in S$ 
Initialize  $s$  arbitrarily
repeat
  For each step:
    Choose action  $a$  based on policy  $\pi(s)$ 
    Get reward  $r$  and next state  $s'$ 
     $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$ 
     $s = s'$ 
until Finished required number of steps or  $s$  reaches a terminal state
```

this algorithm to the previous grid game. We initialize the action probabilities to be equal in all directions; therefore, as in Section 2.7, $\pi(s, a) = 0.25$. We set the discount factor $\gamma = 0.9$, just as we did in the previous example, and set the learning rate to $\alpha = 0.001$. We then run the algorithm for 1,000,000 steps and get the value of the state as

we

$$V^T = [8.89 \quad 2.45 \quad 0.04 \quad 2.56 \quad 0.97 \quad -0.37 \quad -0.13 \quad -0.42 \quad -1.27].$$

Recall that the true value of the state is

$$V^T = [8.85 \quad 2.5 \quad -0.07 \quad 2.5 \quad 0.92 \quad -0.44 \quad -0.07 \quad -0.44 \quad -1.27]$$

There is a major difference in what we are doing in this section and what we did in the case of value iteration. In the case of value iteration we knew the state transition probabilities $P_{ss'}^a$ and the rewards $R_{ss'}^a$, and we simply implemented an iterative search algorithm for the zero of the function but we did not actually play the game. In this section, the agent has to play the game many times to *learn* the value of the states. In the value iteration case, it required only 40 iterations to get an excellent estimate of the value function, and in this section we played the game taking 1,000,000 steps and getting the rewards and then estimating the state value function.

2.10 TD Learning of the State-Action Function

In this section, we estimate the state-action function $Q(s, a)$. We will use essentially the same algorithm as in the previous section. However, in

this case we use an ϵ -greedy action selection process. Therefore, we do not select an action based solely on the current policy, but we choose the next action based on the highest value in the state-action table. This is similar to the n -armed bandit problem. Then with a small probability ϵ we arbitrarily choose another action; this is the exploration phase. The recursive equation is

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha(r_{t+1} + \gamma Q_k(s_{t+1}, a_{t+1}) - Q_k(s_t, a_t)) \quad 2.46$$

We begin the algorithm by initializing the Q-table to random numbers. In our grid game, the Q-table is a 9×4 table. In this case, the algorithm will converge to the optimal Q-table as the algorithm will adapt the action policy to the greedy action. We have set $\alpha = 0.1$ and $\gamma = 0.9$. We run this algorithm for one million steps through the grid. [Table 2.1](#) shows the result.

Algorithm 2.4 Temporal difference state-action algorithm

- 1: Initialize $Q(s, a)$ arbitrarily (random numbers)
 - 2: Initialize the state s arbitrarily
 - 3: Choose action a based on an ϵ - greedy policy
 - 4: **repeat**
 - 5: Take action a and move to the next state, s' , and receive reward r .
 - 6: Choose the next action a' at state s' , using ϵ - greedy policy
 - 7: $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$
 - 8: Set $a = a'$ and $s = s'$
 - 9: **until** specified number of steps complete or s reaches a terminal state.
-

Table 2.1 Temporal difference Q-table learning result.

State	Up	Down	Right	Left
1	22.7	22.6	22.8	22.7
2	17.1	16.0	16.4	20.7
3	15.3	14.1	15.1	18.5
4	20.7	16.4	16.0	17.6
5	18.7	14.4	14.2	18.0
6	16.0	12.8	13.5	16.7
7	18.5	15.1	14.3	15.5

a

be

	15.5	13.7	12.6	16.4
9	14.8	11.6	11.9	13.9

Recall, the value for the *optimal* action policy is

$$V^T = [24.4 \quad 22.0 \quad 19.8 \quad 22.0 \quad 19.8 \quad 17.8 \quad 19.8 \quad 17.8 \quad 16.0]$$

State 1 has the same value regardless of the action taken because from state 1 the agent always goes to state 9 and receives a reward of +10. In this simulation, state 1 was visited nearly 200,000 times and each action s taken approximately 45,000 times and the estimated value of the state-action function is $Q(s, a) \approx 22.7$. If we look at state 2, we find that action *left* has the highest value of $Q(2, left) = 20.7$; therefore, in state 2 the st choice would be to go *left*. We can go down the rest of the table and choose the action that has the highest expected value. As such, an agent playing this grid game can look at its current state and then look up in the Q-table to find which action would give it the highest reward and then, if it were a *rational* agent, it would take the action with the highest reward.

2.11 Q-Learning

This is a well-established reinforcement learning method. It was first proposed in Reference [6] and the proof of stability of this algorithm is sed on stochastic approximation and presented in Reference [7]. The recursive equation for the Q-table takes the form

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t)) \quad .4$$

This algorithm is very similar to the temporal difference (TD) learning algorithm in the previous section. The key difference is that we now search for a strictly greedy policy, without exploration, when we compute $\max_a Q_k(s_{t+1}, a)$. We once again begin by initializing the state arbitrarily and initializing the Q-table $Q(s, a)$ arbitrarily. We choose an action based on an ϵ -greedy action selection policy and then move to the next state s' and get a reward. Then, when we compute the prediction error as $r_{t+1} + \gamma \max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t)$, we do not use an ϵ -greedy algorithm, we search for the maximum value in the Q-table. The algorithm can be itten as Algorithm 2.5.

Algorithm 2.5 Q-learning

Initialize $Q(s, a)$ arbitrarily (random numbers)
Initialize the state s arbitrarily
repeat
 Choose action a based on an ϵ -greedy policy
 Take action a and move to the next state, s' , and receive reward r .
 $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_a Q(s', a') - Q(s, a)]$
 Set $s = s'$
until specified number of steps complete or s reaches a terminal state.

Table 2.2 Temporal difference Q-table learning result.

State	Up	Down	Right	Left
1	23.6	23.7	24.1	23.8
2	18.3	17.2	16.7	21.7
3	16.3	15.0	16.3	19.0
4	21.5	17.2	17.1	18.3
5	19.5	15.4	15.4	18.9
6	16.8	13.6	14.5	17.3
7	19.3	16.4	14.9	16.1
8	17.5	14.3	13.7	16.7
9	15.1	13.0	12.9	15.7

be

Once again, we set $\alpha = 0.1$ and $\gamma = 0.9$ and run this algorithm for one million steps through the grid. We get almost exactly the same Q-table as before. We then run the algorithm again, but this time we set the exploration parameter $\epsilon(k) = \epsilon(0) \div (1 + 0.000001 * k)$, where k is the step number. We then get the results illustrated in (Table 2.2). In this case, the algorithm has converged closer to the optimal values.

2.12 Eligibility Traces

In the previous sections, in the TD learning of the state-value function or the Q-table we used what is known as a *one-step predictor*. The update equations had the form

$$x_{k+1} = x_k + \alpha(r_{t+1} + \gamma y(t+1) - y(t)) \quad 2.48$$

where the TD term $r_{t+1} + \gamma y(t+1) - y(t)$ is based only on what happened in the last time step. In other words, we only updated the effect of the last step. In the case of eligibility traces, we will look back further in time. For example, if the current reward is good, then let us not only update the current state but also give some of the reward to some of the previous states that led us to this point. This should improve the time of convergence of the algorithm. Recall, we are defining the rewards in terms of discounted future rewards as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots + \gamma^{T-t-1} r_{T-1} = r_{t+1} + V_t(s_{t+1}) \quad 2.49$$

A two-step predictor has the form

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2}) \quad 2.50$$

The eligibility trace keeps track of the last time we visited a particular state. The current reward is then assigned to recently visited states. States that have not been visited for a long time are not given much credit for the current reward.

Let us define the eligibility trace for each state at time t as $e_t(s)$. The eligibility trace for each state decays as $\gamma\lambda$, and the eligibility trace for the state just visited is increased by 1. Therefore, we update the eligibility trace as

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \quad 2.51$$

The one-step prediction error is

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \quad 2.52$$

The correction for each state becomes

$$\Delta V_t(s) = \alpha \delta_t e_t(s) \quad \forall s \quad 2.53$$

Algorithm 2.6 TD(λ) learning

```
Initialize  $V(s)$  arbitrarily (random numbers)
Initialize the state  $s$  arbitrarily
Initialize  $e(s) = 0$ 
repeat
  Choose action  $a$  based on  $\pi(s)$ 
  Take action  $a$ , observe reward and move to the next state,  $s'$ 
  Compute TD error as,  $\delta = r + \gamma V(s') - V(s)$ 
  Compute  $e(s) = e(s) + 1$ 
  repeat
    For all states,  $s$ 
       $V(s) = V(s) + \alpha \delta e(s)$ 
       $e(s) = \gamma \lambda e(s)$ 
  until All states update and set  $s = s'$ 
until specified number of steps complete or  $s$  reaches a terminal state
```

We run the TD(λ) described in Algorithm 2.6, and set $\gamma = 0.9$, $\alpha = 0.001$ and $\lambda = 0.9$. We run through the grid for 1,000,000 steps. The estimated value of the value function is then

$$V^T = [8.87 \quad 2.49 \quad -0.23 \quad 2.5 \quad 0.84 \quad -0.49 \quad -0.06 \quad -0.47 \quad -1.29]$$

However, this is almost the same result as we got when we used the traditional TD learning algorithm described in Algorithm 2.3. The promise of using eligibility traces is the increased speed of convergence. Therefore, we run the simulation again and illustrate the convergence of $V(1)$. As illustrated in [Fig. 2-7](#), the curve representing the eligibility trace does converge faster. Whether the improvement in the convergence rate is worth the extra effort and computational requirements is a user's decision. Furthermore, in this particular example, the algorithm is not sensitive to changes in the parameter λ .

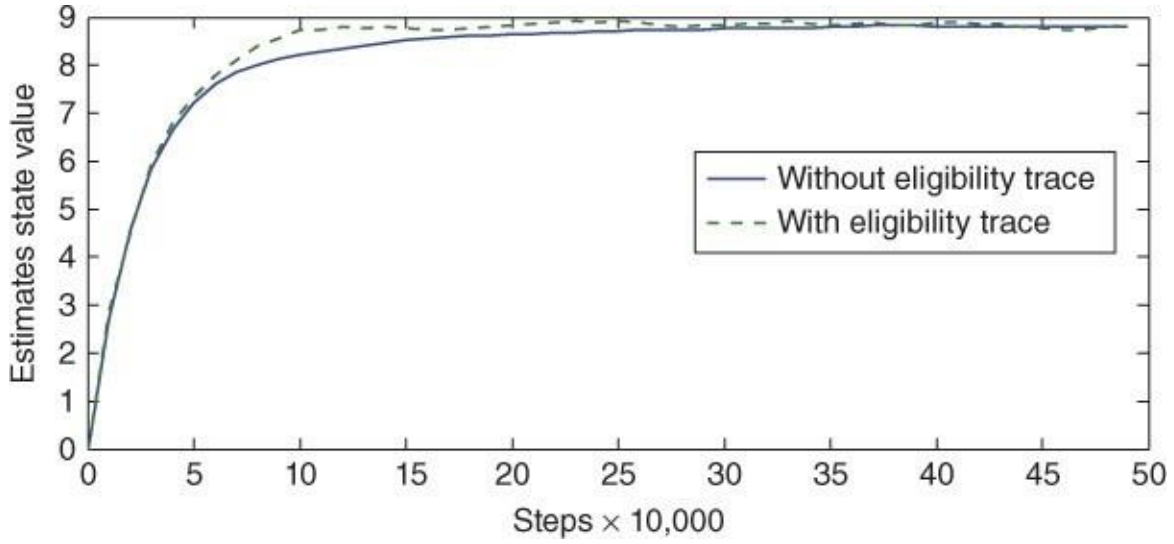


Figure 2-7 Comparison of TD learning with and without eligibility traces for $V(1)$.

The final algorithm in this chapter is the $Q(\lambda)$ learning algorithm. In this case we apply the $TD(\lambda)$ corrections to the state-action Q -table. The eligibility functions then become $e(s, a)$. Everything else is largely the same as in the previous algorithm. As usual, we will compute the prediction error as

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \quad 2.54$$

and the eligibility trace becomes

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma \lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \quad 2.55$$

and then the update for $Q(s, a)$ becomes

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t e_t(s, a) \quad \forall s, a \quad 2.56$$

The algorithm is described in Algorithm 2.7.

Algorithm 2.7 $Q(\lambda)$ learning

Initialize $Q(s, a)$ arbitrarily (random numbers)
Initialize the state s arbitrarily
Initialize $e(s, a) = 0$
repeat
 For each step
 Choose action a based on $\pi(s)$
 Take action a , observe reward r and move to the next state, s'
 Choose a' from s' using an ϵ -greedy policy.
 Compute TD error as, $\delta = r + \gamma Q(s', a') - Q(s, a)$
 Compute $e(s, a) = e(s, a) + \delta$
 repeat
 For all states, s and a
 $Q(s, a) = Q(s, a) + \alpha \delta e(s, a)$
 $e(s, a) = \gamma \lambda e(s, a)$
 until All states update and set $s = s'$ and $a = a'$
until specified number of steps complete or s reaches a terminal state

Setting $\lambda = 0.9$ and $\gamma = 0.9$ and $\epsilon = 0.1$ and $\alpha = 0.1$, we get almost the same answer as with the standard Q -learning as shown in [Table 2.2](#). Once again, we examine the rate of convergence for the first value in the table, namely $Q(1, \text{UP})$. The graphs of the convergence of the term $Q(1, \text{UP})$ is illustrated in [Fig. 2-8](#). The difference between the convergence rates for the Q -learning algorithm and the $Q(\lambda)$ -learning algorithm is relatively minor in this particular case. One has to use good judgement to determine whether using the more complex $Q(\lambda)$ -learning algorithm is of significant value.

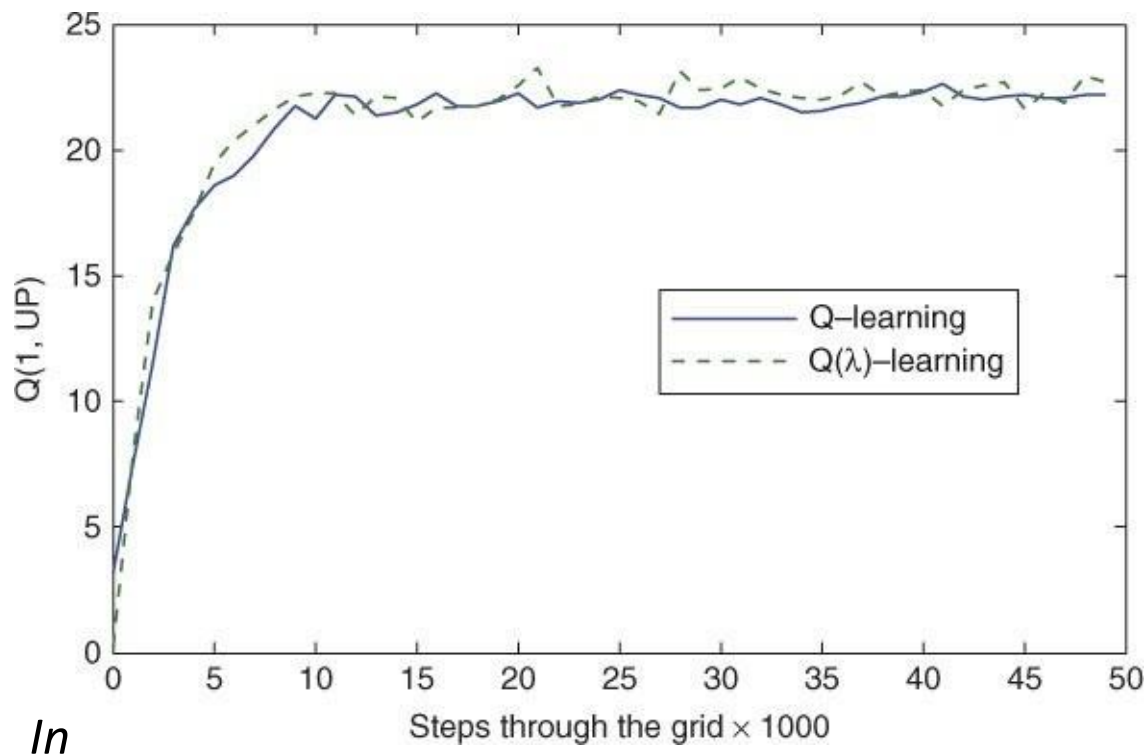


Figure 2-8 Comparison of Q-learning with and without eligibility traces for $Q(1, UP)$.

References

- [1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: The MIT Press, 1998.
- [2] M. Bowling and M. Veloso, "Multiagent learning using a variable learning rate," *Artificial Intelligence*, vol. **136**, no. 2, pp. 215–250, 2002.
- [3] R. Bellman, *Dynamic Programming*. Princeton, New Jersey: Princeton University Press, 1957.
- [4] D. P. Bertsekas, *Dynamic Programming: Deterministic and Stochastic Models*. Englewood Cliffs, New Jersey: Prentice-Hall, 1987.
- [5] R. A. Howard, *Dynamic Programming and Markov Processes*. Cambridge, Massachusetts: MIT Press, 1960.
- [6] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. **8**, no. 3, pp. 279–292, 1992.

[7] T. Jaakkola, M. Jordan, and S. Singh, “On the convergence of stochastic iterative dynamic programming algorithms,” *Neural Computation*, vol. 6, no. 6, pp. 1185–1201, 1994.

2. Literature Review

CHAPTER 1

Best Practices in Mission-Assured, Mission-Critical, and Safety-Critical Systems

Kim Fowler

1. Roadmap to This Book

The material in this book presents best practices in developing mission-assured, mission-critical, and safety-critical systems for medical devices, avionics, military equipment, and spacecraft subsystems. It is a disparate grouping. The good news is that there are threads of commonality in best practices among these systems. The acronyms may vary but best practices translate fairly easily from one area to another. The following are selected areas where best practices in one market often translate well into other markets:

- Defined processes, procedures, and quality assurance (QA) programs
- Collecting and managing requirements and setting priorities for their implementation
- Experimentation and prototyping
- Risk assessment—technological feasibility and business feasibility
- Rigorous software development processes
- Review and inspection
- Documentation
- Test planning, verification and validation, and acceptance testing

1.1. *Systems Engineering*

The material in this book takes a high-level architectural approach. In essence, it is systems engineering. Whether medical, avionics, military, or space, all deal with multiple

disciplines, including (but not limited to) software, electronic hardware, mechanical subsystems, and operations. The systems engineering approach pulls these disciplines together into a framework that helps elucidate interactions between disciplines. Basic definitions that you will need for the remainder of the book follow:

System: A combination of elements or parts forming a complex or unitary whole; composed of components, attributes, and relationships. Typically these elements within a system form definable inputs, processing, and outputs. The interrelated components work together toward a common objective [1].

Systems engineering: An “engineering discipline whose responsibility is creating and executing an interdisciplinary process to ensure that the customer and stakeholder’s needs are satisfied in a high quality, trustworthy, cost efficient, and schedule compliant manner throughout a system’s entire life cycle. This process is usually comprised of the following seven tasks: State the problem, Investigate alternatives, Model the system, Integrate, Launch the system, Assess performance, and Re-evaluate.. . . [The] [s]ystems [e]ngineering [p]rocess is not sequential. The functions are performed in a parallel and iterative manner” [2]. Brian Mar states that systems engineers should adhere to the following basic core concepts:

- Understand the whole problem before you try to solve it
- Translate the problem into measurable requirements
- Examine all feasible alternatives before selecting a solution
- Make sure you consider the total system life cycle
- Test the total system before delivering it
- Document everything [2]

Everyone on the team should exercise the discipline of systems engineering. The leader of the effort, aside from the program manager, should be a systems engineer or a systems architect. The difference between engineering and architecting is that engineering is deductive work based on hard science, while architecting is inductive work that tends to be likened to art [3]. Both are important in systems engineering.

Mission assurance: “Mission [a]ssurance is a full life-cycle engineering process to

identify and mitigate design, production, test, and field support deficiencies of mission success” [4].

“Mission [a]ssurance includes the disciplined application of system engineering, risk management, quality, and management principles to achieve success of a design,

development, testing, deployment, and operations process. Mission [a]ssurance's ideal is achieving 100% customer success every time. Mission [a]ssurance reaches across the enterprise, supply base, business partners, and customer base to enable customer success" [5].

"Mission-[a]ssured [d]esign is aimed at creating a robust, manufacturable system—one that does exactly what it is intended to do (its mission) despite the presence of variations, stresses, and uncertainties that can result in mission failure" [6].

Mission critical: "The term mission critical . . . refers to any factor (equipment, process, procedure, software, etc.) . . . [that] is crucial to the successful completion of an entire project. It may also refer to a project the success of which is vital to the mission of the organization which attempts it" [7].

Safety-critical systems: "A computer, electronic or electromechanical system whose failure may cause injury or death to human beings. . . . [Examples include] an aircraft or nuclear power station control system. Common tools used in the design of safety-critical systems are redundancy and formal methods" [8].

Quality: The degree for which the sum total of product characteristics fulfill all of the requirements of customers.

Process: A group of interrelated activities and resources that transforms inputs into outputs, often described by a block or flow diagram of events.

Procedure: Specific implementation of the process for a single, focused area of concern; typically step-by-step instructions.

Validation: The confirmation that the design, function, and operation of the final product satisfies the customer's intent.

Verification: The objective tests of metrics that show that the final product meets the quantitative requirements.

1.2. Important Issues

Five basic issues affect every development:

- Integrity—Development requires an integrated approach.
- Interfaces—Most important actions occur at interfaces.
- Humanity—All problems have a human origin.

- Iteration—All levels of development should have feedback to revise direction and design.
- Multidimensions—There are no silver bullets for completing a project.

Integrity: An integrated approach requires integrity, which is defined as the “seamless whole.” This requires a “big-picture view” of how the parts fit into the whole. The project lead, in particular, needs to maintain this vision of development and needs to communicate it to the team. Doing so also requires a balanced view of concerns and a willingness to listen to differing opinions from people in different disciplines. This will build trust and a more effective team. A colleague and friend of mine, Jeff Sieracki, occasionally will comment on the need to understand the integration of different disciplines within a project’s development. Jeff will say somewhat wryly, “If it’s not in my field of expertise, then it must be trivial.” Jeff’s point is that this colloquial attitude needs to be jettisoned if a project is to truly succeed.

Interfaces: All, or nearly all, important actions occur at interfaces. Think about it, most interesting things happen where two different domains meet. Wave propagation, for example, doesn’t get interesting until it encounters an obstacle; then diffraction, refraction, and reflection take place. Semiconductor operations occur at the boundary of different materials. Even the interface between human operations and machine functions is interesting and important. (The one place this view weakens is in software— but even then mistakes like unprotected underflow or overflow or illegal operations are a sort of “boundary.”)

One part of successful development is to understand the various types of interfaces (e.g., electrical, material, mechanical, software, signal flow, environmental, human–machine interactions). This requires defining the boundaries and subsystems and then using these to partition the development.

Humanity: All problems have a human origin. Even the very best designs will fail eventually or outlive their usefulness. Human design cannot account for all possibilities—circumstances that were completely unexpected or unknown, multiple and simultaneous interactions that lead to failure, and even human abuse from inappropriate, stupid, or malicious operations. Beyond these concerns, the finite lifetime will limit normal use over time. Failure, furthermore, can have other causes, such as business climate, socioeconomic factors, or politics, that defy technical solution.

Iteration: Development needs a closed-loop, feedback form of operation that includes planning, execution, and

review. Discussion of this is addressed later in the section covering the PERRU (plan, execute, review, report, and update) concept.

www.newnespress.com

Multidimensions: Every interesting problem that needs solution requires the involvement of diverse disciplines. There are no “silver bullets.” Simply no tool or method or process fits all operations. Don’t look for one. Any “silver bullet” will narrow your focus and you will overlook important issues and concerns.

1.3. *Material Covered*

The material in this book covers best practices, guidelines, government regulations, and standards in developing mission assurance with mission-critical and safety-critical equipment. With these concerns, the book addresses the following disciplines within product and system development:

- Systems
- Operations research
- Design and development
- Software
- Electronic hardware
- Mechanics
- Human interface
- Testing
- Fielding

Areas that are important but not addressed in detail in this book are information technology (IT), manufacturing, distribution, marketing, logistics and support, legal, and business and accounting.

Medical devices, military equipment, and spacecraft instruments share common concerns in many areas, even though at first blush they appear to be very different and operate in very different environments. Common concerns include mission assurance, reliability, availability, vibration, shock, rigorous development, and extensive testing. In particular, mission assurance with mission-critical and safety-critical systems does not and should not rely on any single discipline to ensure performance.

The main difference between areas is that emphasis may change in usage, environment, software rigor, change or update frequency, dependability, and maintenance. [Table 1.1](#) gives examples of some differences in emphasis; these examples only point to the majority of situations, obviously exceptions exist.

Table 1.1: Comparison of Differences of Emphases by Application Type

	Medical Devices	Military Equipment	Spacecraft Instruments
Human interface	Implanted: none External: simple and very clear	Can be fairly complex requiring considerable training of users	Instrument: none GSE: can be complex but only a very few “gurus” will run it; much less effort needed to build intuitive interfaces; interface needs to allow extensive control of system
Environment	Implanted: body temperature, minimal shock, low vibration, corrosive fluids External: office ambient, 0 to 340°C, shock of 1-m drop to concrete floor, minor vibration	COTS: office ambient, 0 to 340°C, shock of 1-m drop to concrete floor, minor vibration Field: -40 to 380°C, wide variety of shock and vibrations	Instrument: -40 to 380°C, shock and vibration of launch GSE: office ambient, 0 to 340°C, shock of 1-m drop to concrete floor, minor vibration
Software development rigor	Class III devices that have life-sustaining operations require certified compilers and RTOS, and very rigorous development and review by FDA	COTS: none Field: rigorous development, depends on review and oversight by customer, some guidelines and standards	Instrument: rigorous development, depends on review and oversight by customer, some guidelines and standards GSE: only review required by customer
Update frequency and type	None, once approved by FDA; requires completely new approval cycle for changes and updates	Depends on architecture of equipment; line replaceable units (LRUs) allow changes quite readily	Instrument: usually none unless minor patches transmitted up GSE: as required
Certification	FDA approval	None unless UL or CE required by customer	None
Reliability	Implanted: a top priority External: some importance	Important, particularly for field applications where repair facilities are not available	Instrument: a top priority GSE: no requirement for COTS usually
Availability, diagnostics, and repair	Not important—devices replaced, not repaired in patient or field	Very important for repair	Instrument: none GSE: no requirement for COTS usually
Safety	Highest priority	Usually a matter of training, not of design requirements	Instrument: none, except for assembly GSE: none

Table 1.1: Cont'd

	Medical Devices	Military Equipment	Spacecraft Instruments
Maintenance	Either none or personnel trained to perform replenishment and logistics	Very important in both design requirements and in training of operating personnel	Instrument: none GSE: none
Disposal	FDA requirement for supplier to track, record, and dispose	As required by customer or by international standards (i.e., RoHS and WEEE)	Instrument: none GSE: RoHS and WEEE

Notes: Specific conditions and parameters are for examples only. Considerations for spacecraft instruments pertain to unmanned spacecraft only.

GSE, ground support equipment; RoHS, restriction of use of certain hazardous substances, which requires the phase-out of lead, mercury, cadmium, chromium, halogen, bromide, and some fire retardants; WEEE, waste from electrical and electronic equipment, which requires the treatment, recovery, and recycling of electric and electronic waste.

Source: # 2009 by Kim Fowler. Used with permission. All rights reserved.

2. Best Practices

2.1. *What and Why?*

Best practices are processes, procedures, methods, and techniques that help development of good-quality products. These best practices have to do with what works well in conceiving, planning, architecting, analyzing, designing, building, and testing mission- critical and safety-critical systems.

Best practices, properly applied, will improve the probability of success; they will not guarantee success. Interestingly, best practices in one field or market arena often have analogs in other fields or markets.

Since the 1970s while attempting to achieve “better, faster, cheaper” development of quality products, companies in most industries have acceded that good quality is a necessary ingredient to profitability and even survivability. Best practices include aspects of implementing and operating a quality system. A quality management system (QMS) should blanket a company, which means that it affects every area as a philosophy that guides operations. It should never be an “added management structure” for looking over people’s shoulders. The most important realization of a QMS is that it becomes an individual responsibility, as well as a corporate mandate.

2.2. *Rationale*

A primary goal of a QMS is mission assurance—delivering the right product to the right people in the right application and operating as expected. Toward that goal, consistent application of the QMS across the enterprise is important, which means instituting

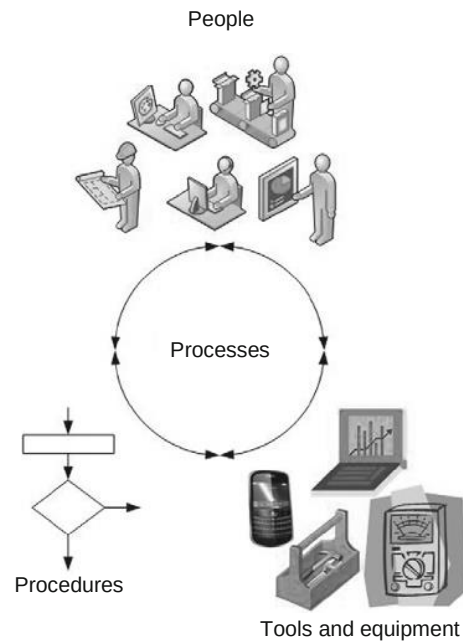


Figure 1.1: Processes link people, tools, and procedures to support consistent application of effort toward quality across the enterprise. (© 2008–2009 by Kim Fowler.

Used with permission. All rights reserved.)

appropriate processes that involve people, tools of industry and business, and procedures (Fig. 1.1). The most important part of QMS is the people. The wrong people—incompetent or unmotivated or worse—will undermine and destroy all other efforts. From this point on, the book will assume that the right people are already in place. The right people can make anything work; good processes tying together people, tools, and procedures, will help them be more effective and efficient.

2.3. *Standards and Guidelines for a QMS*

2.3.1. *QMS Components*

The QMS has the following requirements: It identifies processes, controls processes, documents the effort through record keeping, and controls the records. Regardless of their formats, all quality systems should have the following primary components:

- Implementation
- Management and staff responsibility
- Resource management

- Product (or service) realization
- Measurement, analysis, and improvement

2.3.2. Responsibility within QMS

Usually a quality manager will establish policy to cover the following concerns:

- Customer focus
- Quality policy
- Planning
- Authority
- Communications
- Review

These areas tend to be the most visible; staff and customers often associate these with the QMS. Ultimately a quality system and its best practices within a company should be everyone's responsibility. [Table 1.2](#) lists the people for establishing the QMS and leading the infusion of best practices into your company.

2.3.3. Resource Management within QMS

Resource managers within the QMS must deal with the following components:

- Human competence, awareness, and training
- Communications and data flow between customer, business areas, and staff
- Infrastructure, including work space, utilities, and equipment
- Environment, including cleanliness, safety, and sound levels

Table 1.2: People Who Can Establish the QMS and Implement Best Practices within a Company

Company Department/Activity	Responsible Leader
Business	CEO, CFO, company president, or business VP
Engineering	CTO, engineering VP, or an engineering manager
Product and quality assurance	CEO, quality manager, or quality lead
Manufacturing and production	Manufacturing VP or manufacturing manager
Personnel	Human resources manager
Infrastructure	CEO, COO, or plant manager

2.3.4. Product Realization within QMS

Product realization within the QMS includes the following components:

- Planning
- Customer input and requirements
- Design and development
- Purchasing
- Production and manufacturing
- Service and support
- Control of measuring equipment

2.3.5. Development and Certification of the QMS

Four types of quality processes are prominent in many industries: ISO 9001, AS9100, Six Sigma, and CMMI. Choosing one depends on its fit to the industry and the understanding of the particular QMS by both employees and customers. Brief descriptions of each type follow.

ISO 9001

ISO is the acronym for International Organization for Standardization, which is based in Geneva, Switzerland. ISO 9001 provides a quality management system defined as, “a series of components logically linked together that provides measures and controls to manage and improve products” [1]. To be ISO certified, a company needs to purchase the standard and then be regularly audited.

ISO 9001 comprises two basic programs—one provides structure and the other provides for improvement. The structure program accomplishes the following:

- Identifies processes and documents them
- Describes sequences and interactions
- Ensures resources to run the quality management

system The improvement program provides the following:

- Measures performance
- Judges effectiveness
- Improves the quality management system [9]

ISO 9001 requires an eight-section QMS. The most prominent are the last five. [Section 4](#) is about documentation, particularly the quality manual. [Section 5](#) concerns management responsibility, and [Section 6](#), resource management. [Section 7](#) is about product realization, and [Section 8](#), measurement, analysis, and improvement.

AS9100

AS9100 is defined specifically by and for the aerospace industry [10]. It was developed from AS-9000 (1997), which adopted the 20 elements of ISO-9001-1994, and aligned them with the needs of the air transport industry. The AS9100 incorporates additional provisions to cover civil and military aviation and aerospace industry standard requirements more comprehensively. It also covers the needs of suppliers, regulatory bodies, and customers of the aerospace industry.

AS9100 has the following objectives:

- Reduce defects in the supplier chain
- Reduce costs
- Continuously improve quality
- Enhance customer satisfaction
- Reduce the person-hours spent by an organization on quality management activities, primarily by replacing the various customer requirements and specifications with a single industry standard that's acceptable to everyone

Key features of AS9100 include:

- Quality improvement
- Variation control or management for key product characteristics
- Production and service provisions unique to the aerospace industry such as part accountability, foreign object detection, production documentation, part identification, and part traceability
- Process/tooling change control and management
- Supply-chain quality control, which covers the control of purchasing and acceptance processes
- Design and development control
- Product configuration control management

- Product quality, reliability, and safety control
- Continual improvement

Some AS9100 definitions are as follows:

- Product (AS9100 service)—Activity that adds value for a customer. In aerospace this might be transporting cargo.
- Product (AS9100 software)—Computer program that adds value for a customer.
- Product (AS9100 hardware)—Physical entity that adds value for a customer. A hardware product might be a mechanical engine or motor or it might be an electronic circuit board or it might be a mechanism or a structural enclosure.
- Product (AS9100 processed material)—Substance requiring replenishing that adds value for a customer. One example is lubricant.
- Authority (AS9100)—National aviation authority having jurisdiction over activities affecting the airframe.
- Key characteristic (AS9100)—Features of a material, process, or part whose variation has a significant influence on product life, performance, service life, or manufacturability.

Six Sigma

The Six Sigma quality system originated in high-volume production and manufacturing. Its main goal is to identify and measure variances. Six Sigma does not guarantee quality but provides expectations of program performance based on customer satisfaction [11].

CMMI

CMMI is the acronym for Capability Maturity Model Integration. It grew out of development guidelines for software through the Software Engineering Institute at Carnegie Mellon University. The guidelines have unlimited distribution rights and can be downloaded from www.sei.cmu.edu.

CMMI has five levels of maturity that represent stages and capability of a company with its processes. Table 1.3 lists and briefly describes the levels of maturity.

CMMI has 22 process areas (PAs); each PA is a cluster of related best practices to satisfy goals to improve processes. Table 1.4 lists the PAs along with the maturity level for a staged representation.

Table 1.3: Maturity Levels in CMMI for Staged Representation

Level	Name	Comments
1	Initial	Success depends on competence and heroics.
2	Managed	Processes are planned and executed according to policy.
3	Defined	Processes are tailored to each project and described more rigorously.
4	Quantitatively managed	Quantitative objectives are measured and demonstrate predictable performance.
5	Optimizing	Continually improves processes based on quantitative understanding.

Table 1.4: General Areas and PAs for Stage Representation of CMMI

Areas	Maturity Level
Requirements management Project planning Project monitoring and control Supplier agreement management Measurement and analysis Process and product quality assurance Configuration management	2
Requirements development Technical solution Product integration Verification Validation Organizational training Integrated project management Risk management Integrated teaming Integrated supplier management Decision analysis and resolution Organizational environment for integration	3
Organizational process performance Quantitative project management	4
Organizational innovation and deployment Causal analysis and resolution	5

Selected CMMI definitions follow:

- Process area (PA)—Cluster of related practices in an area that, when implemented collectively, satisfy a set of goals considered important for making improvement in that area.
- Generic goal (GG)—Goal that applies to multiple PAs and describes the characteristics to institutionalize processes; it is a required model component and is used in appraisals to help determine whether a PA is satisfied.
- Specific goal (SG)—Goal that has unique characteristics and satisfies a specific PA; it is a required model component and is used in appraisals to help determine whether a PA is satisfied.
- Generic practice (GP)—Describes an activity that is important for achieving the associated generic goal; it is an expected model component.
- Specific practice (SP)—Description of an activity that is important for achieving the associated specific goal; it is an expected model component.

Comparison of ISO 9001 versus CMMI

ISO 9001 tends to be more of a “pass-or-fail” sort of effort. You describe a company’s processes and then show how the company meets and performs these processes. It does not provide as many guidelines or details to incorporate into processes.

CMMI is more “granular” or has higher resolution than ISO 9001. CMMI provides more detailed guidelines than ISO 9001. The intent of CMMI is to focus on process improvement through assessment and maturity levels. [Table 1.5](#) provides a comparison between ISO 9001 and CMMI.

Table 1.5: ISO 9001 versus CMMI

	ISO 9001	CMMI
Characteristics	Stereotyped; “pass-fail” test	More flexible; focus on assessment and improvement
Advantages	Easily adaptable to manufacturing Customers understand	Adapts well to software Easy-to-get standard
Disadvantages	Less direction for improvement Cost to undergo audits	Learning curve for the 22 PAs Must train customers to understand

CMMI Framework CMMI attempts to prevent sole focus on a specific business area. It transcends disciplines and includes software, hardware, mechanical, logistics, service, and maintenance. CMMI incorporates best practices for both development and maintenance. Watts Humphrey outlines five ideas that inspired the goals of CMMI:

- Planning, tracking, and schedule management
- Requirements definition and configuration control
- Process assessment
- Quality measurement and continuous improvement
- Evolutionary improvement [12]

Table 1.6 gives the stages of implementation and the PAs implemented in each stage. **Appendix A** contains a description of each PA.

2.3.6. Training for QMS

Implementing a quality system and best practices within a company requires training all staff and personnel. The goal is to make the quality system useful and to make the training efficient and appropriate. Consequently, training should occur at several stages and levels. For example, a company might do the following:

- Short, weekly session of about 20 minutes
- Monthly session of about 1 hour
- Several days every 3 months to evaluate the processes, suggest changes, and train for more detail in the current stage or prepare for the next stage

The most important concern is for training to be regular, appropriate, effective, and inclusive.

2.3.7. Measurement, Analysis, and Improvement within QMS

Measurement, analysis, and improvement within the QMS have the following goals and components:

- Ensure product, service, and QMS uniformity
- Customer satisfaction
- Internal audits
- Monitor processes
- Control of nonconforming product

Table 1.6: CMMI PAs Established at Each Stage of Implementation

Level	Components Implemented	PA	CMMI Level
1	Workorder (WO), Problem report corrective action (PRCA), requirements development	Project planning Requirements development Requirements management Measurement and analysis Process and product quality assurance	2 3 2 2 2
2	Engineering processes and checklists, configuration management, document templates	Project monitoring and control Technical solution Supplier agreement management Configuration management Organizational training Integrated project management	2 3 2 2 3 3
3	Engineering updates, manufacturing updates, vendor qualification and management	Product integration Verification Validation Risk management Decision analysis and resolution Organizational environment for integration	3 3 3 3 3 3
4	R&D, project processes integrated	Integrated teaming Integrated supplier management Organizational process performance Quantitative project management	3 3 4 4
5	Business processes integrated	Organizational innovation and deployment Causal analysis and resolution	5 5

- Analyze data
- Improvement (continuous operation, corrective action, preventive action)

Measurements, analyses, and updates improve the quality of the process and procedures. The planning block establishes the metrics for measurement; these metrics are a part of the requirements. Every requirement should have a metric or measurable/observable quantity to allow later verification and validation.

Once the measurements are made, the data are analyzed for methods of improvement, which happens during the review stage or block. The updates then implement the improvements that derive from the review.

The review and report are both qualitative and quantitative. Learning derives from the update activities.

3. Project Management and Systems Engineering

There is considerable overlap between project management and systems engineering—both are leadership positions and both must deal with many different people and disciplines while maintaining budget, schedule, and output quality. The difference between project management and systems engineering is small; project management is more business oriented while systems engineering is more technical.

For smaller projects, typically 20 or fewer people, one person usually occupies both positions and leads the projects. For large-scale system development, such as a vehicle or facility design, the team may have multiple levels of both project managers and systems engineers. For these large developments, a systems architect may also join the team and assist system engineers and team leaders with the technical architecture of the system.

3.1. *Project Management*

Managing a project has several component activities. Project management establishes the contract for the project and selects the type of development. It determines the team and subcontractors who will complete the project. Project management sets the schedule and budget. It controls the project through task or work orders and then monitors the project through the completion of task/work orders, various reviews, and the control board. Finally, project management closes out the project.

Verzuh states, “The ultimate challenge for project managers is to meet the cost, schedule, and quality of the project without damage to the people. That means the project ends with high morale, great relationships with customers, and vendors [who] can’t wait to work with you on the next project” [13].

3.1.1. *Definitions*

Project: Development work that has a clear beginning and end and is performed once to produce something unique. This work is different from ongoing operations such as manufacturing [13].

Project management: Verzuh claims that project management is “art informed by science.” He lists five characteristics of a successful project management:

1. Agreement among the project team, customers, and management on the goals of the project.
2. A plan that shows an overall path and clear responsibilities and will be used to measure progress during the project.
3. Constant, effective communications among everyone involved in the project.
4. A controlled scope.
5. Management support [13].

3.1.2. Inputs and Outputs

The inputs to project management are the customer’s expressed intent and requirements, project constraints, market conditions, standards, guidelines, and the contract. Outputs are descriptive documents, reviews and reports, and physical product.

3.1.3. Operations

In between the inputs and outputs are the operations. Operations include the following areas:

- Concept and market
- People and disciplines
- Architecting, architecture, and process flows
- Scheduling
- Documentation
- Requirements and standards
- Analyses
- Design tradeoffs
- Test and integration
- Manufacturing
- Support and service

Project management has the following functions:

Definition—Determine the purpose, goals, and constraints for the project, and set the rules and guidelines.

Enlist people—Get a sponsor and determine the stakeholders for the life cycle of the project.

Communicate—Prepare and deliver the statement of work, the responsibility matrix, the charter of authority, and the communication plan.

Planning—Estimate effort and cost, prepare the budget and schedule, define the risk.

Control—Measure progress, manage risk, ensure that corrective actions take place, hold reviews, and maintain communications [13].

3.1.4. Best Practices—Scheduling

The project manager leads planning and scheduling but, in general, planning and scheduling are shared responsibilities with the team. It takes a committed team to plan well, work the plan, stick to the plan, and learn from their mistakes. Conversely, failing to plan properly often results in failed schedules.

You can schedule a project several different ways; the most typical are top-down scheduling and bottom-up scheduling. Top-down scheduling sets delivery dates and milestones first and then determines the development that can fit into the periods between dates. Bottom-up scheduling starts with the work break-down structure, which contains the basic details of everything that needs to be done, and builds up to the dates when stages can be finished.

While top-down planning has the overall perspective needed to complete a project, it can over-constrain a project by planning to do too much in too short a time. While bottom-up planning can be very detailed, inappropriately estimated margins and tolerances in timing can build inefficiencies into the process and extend the schedule too far. Use both methods in combination to develop an appropriate schedule. Start with a top-down plan and then see where a bottom-up plan meets it. If they are misaligned, iterate the exercise by relaxing constraints in top-down and tightening tolerances that may be too loose in bottom-up. This kind of exercise can help you identify and remove the pitfalls of either method.

When planning and scheduling, you must first understand the activities that make up project development—the work break-down structure is a great tool here. Remember to include meetings, desk work, lab work, communications, documentation, travel, debugging, testing, fabrication, support, installation, and training, along with the obvious activities of design, production, and big customer reviews.

Think carefully about these things. A simple meeting of several people requires more time than that used to sit around the table and talk. It includes preparation of notes

and slides, communicating and scheduling, and even the very simple act of walking to and from the location. Simple and short telephone calls can fill a day. I have found that calling a vendor and inquiring about the price and delivery on a single, generic, stocked item will average about 8 minutes. Simple activities can eat up time!

www.newnespress.com

Finally, build in a schedule margin for “surprises.” Study where problems have occurred in previous projects and find correlations between the activities and the results. Once you better understand the trouble spots, then you can apportion margin to those activities.

3.2. Systems Engineering

3.2.1. Definitions

Systems engineering: An interdisciplinary approach encompassing the entire technical effort to evolve and verify an integrated and life-cycle balanced set of system, people, product, and process solutions that satisfy customer needs. Systems engineering encompasses the following:

- Technical efforts related to the development, manufacturing, verification, deployment, operations, support, disposal of, and user training for, system products and processes
- Definition and management of the system configuration
- Translation of the system definition into work break-down structures
- Development of information for management decision making [14]

Work break-down structure (WBS): A tool for breaking down a project into its component tasks to initiate, execute, and complete the development of the project. Each task is an elemental, self-contained effort that has clear inputs and outputs and a description of the work. The WBS helps provide a detailed outline of the project scope, monitors progress, contributes to the estimation of cost and schedule, and designates the project team [13].

For military projects, MIL-STD-499B described WBS as “[a] product-oriented family tree composed of hardware, software, services, data, and facilities which result from systems engineering efforts during the development and production of a defense materiel item, and which completely defines the program. Displays and defines the product(s) to be developed or produced, and relates the elements of work to be accomplished to each other and to the end product” [14].

Summary tasks: A higher-level description of a group of tasks that completes a subset of the project or its process. Summary tasks may contain lower-level (more detailed) summary tasks or work packages or both.

Work packages: The most detailed, elemental task that clearly defines a single effort, such as analyzing a simple circuit design or performing a code review of a single software module. Often a work package will be completed in less than 80 hours of effort.

3.2.2. Inputs and Outputs

Inputs and output for systems engineering are very similar to those in project management. The focus tends to be more technical in nature and the requirement metrics more objectively measurable.

3.2.3. Operations

Operations for systems engineering cover the same areas listed above in the discussion of project management. The technical detail is greater for systems engineering and more time is spent on developing the architecture, and confirming, verifying, and validating the work product with the requirements. Systems engineering also focuses on the interactions of components and subsystems within the system. Often the systems engineer will develop the WBS, or a portion of it, for use by the program manager.

3.2.4. Best Practices—Requirements

Requirements form a list of descriptions that define and constrain system design. They specify system behavior and operational parameters, provide a reference for modifications, and characterize potential responses to unexpected situations. A requirement may have one of several different forms, “from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification” [15]. Oshana writes, “A good set of requirements has the following characteristics:

- *Correct.* Meets the need.
- *Unambiguous.* Only one possible interpretation.
- *Complete.* Covers all the requirements.
- *Consistent.* No conflicts between requirements.
- *Ranked for importance.*
- *Verifiable.* A test case can be written.
- *Traceable.* Referring to requirements is easy.
- *Modifiable.* Adding new requirements is easy [15].

Establishing the requirements for a project has several stages, each with associated best practices. The stages follow:

- Gathering
- Establishing

- Applying
- Refining and updating

Gathering: You need direct contact with customers and potential customers to gather requirements. The problem is to understand what they really need. (Note the term “need,” not “want”—there is a significant difference. People often will tell you what they want but don’t always know the extent of their true need.) Your visits to customers are very important; these meetings succeed when discussions between designers and users have prepared templates of guided scripts. A variation on this theme is the use of focus groups, which provide a controlled environment and trained facilitators; focus groups can elicit ideas and directions from potential customers that may not have occurred to you.

Another primary area for gathering requirements is to find and understand the necessary standards and regulations within the particular market. These can often be found online.

Establishing: Once you have begun gathering information for a potential new project, then you can begin developing requirements. Realize that requirements are the codification of intent, which is the desire that a product have particular characteristics and an appropriate function and performance. The trick is to translate intent into requirements, which define and set measurable quantities for later testing, verification, and validation.

Applying: Requirements direct both design and evaluation. The requirements define the metrics of the design. Evaluating the metrics indicates project progress; appropriate requirements will provide a good and complete picture of the progress.

Refining and updating: No list of requirements is ever complete or correct in the beginning. Requirements need to be analyzed through various means: simulation, trade studies, risk analysis, laboratory bench tests, field trials, and discussions with clients and customers.

Later as the project develops, constraints and contradictory requirements will appear—they always do. The list of requirements must be refined and updated to represent the current reality. Prepare for requirements to be flexible and plan for cycles of updates to the requirements list.

3.2.5. Best Practices—Efficient Development of Requirements

The process of developing requirements is an inexact science. Unforeseen problems and unpredictable interactions often arise because the requirements are either too sparse or too complex. Sparse requirements leave out important concerns leading to unexpected problems. Complex requirements can “overspecify” a product leading to unforeseen interactions that cause problems. Reinertsen reports that many projects fail

because requirements are too complete, too stable, and too accurate. These types of requirements shift attention from the few, critical specifications to the many, unimportant ones [16].

www.newnespress.com

Here are some ways to establish useful requirements:

- Write the catalog page for the product first. If a feature warrants mention, then it is important. Any feature that is left out is deemed less important. This will help you prioritize requirements [16].
- “Brief specifications almost always force higher levels of customer contact” [16]. More customer contact leads to a deeper understanding of customer needs, quicker resolution of interpretations, and faster reaction to market shifts.
- High-quality requirements make for concise specifications. The converse is not true—short specifications do not necessarily lead to good quality specifications [17].
- Allow flexibility in the requirements and make development a “closed-loop system.” Reinertsen claims that initial accuracy is not as important as assuming that the specifications have inaccuracies and then using feedback to detect and correct them [16].
- Pazemenas argues for shortening “the fuzzy front end” [18]. Identify new product opportunities and improvements in technology often—quarterly evaluations may be necessary—so that the fuzzy front end does not drag out from a long learning curve.

Some companies are using or moving toward model-based specifications for subsystems to capture requirements. Model-based specifications use a model of the device or system with mathematical constructs, such as sets and sequences, and define system operations by how they modify the system state. Use cases and UML are an example of model-based approaches; these can reveal interactions between the system and external inputs and can force you to consider obscure sequences [15].

Another potential best practice in developing requirements is test-driven development. “Test-driven development (TDD) . . . uses short development iterations based on pre-written test cases that define desired improvements or new functions. Each iteration produces code [or requirements] necessary to pass that iteration’s tests” [19].

TDD originated as a software design technique but serves well for system requirements as well (Fig. 1.2). “We employed TDD to test core behavior and used the best practices of object-oriented design to isolate areas of uncertainty. Work on requirements continued in parallel, reducing the uncertainty in the process. After six months of development, we had only two or three bugs that required more than a few minutes of debugging. The total time spent debugging was only about a week over the six-month project!” [20]

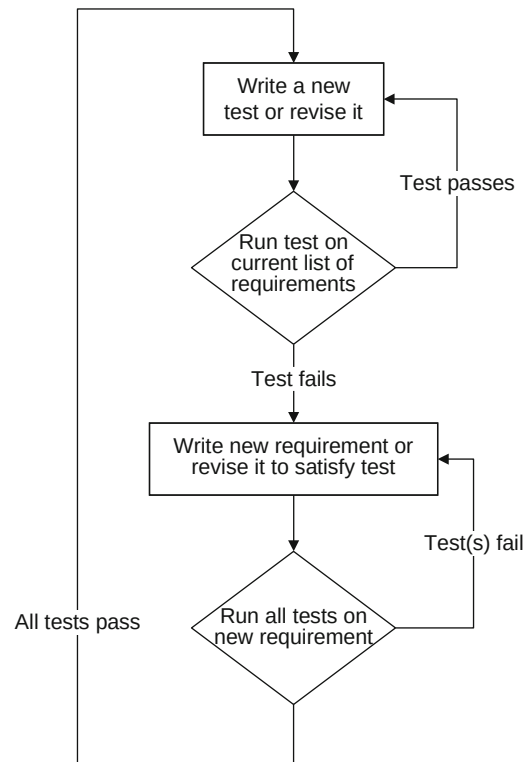


Figure 1.2: Generalized flow for developing requirements through TDD. (© 2009 by Kim Fowler. Used with permission. All rights reserved.)

3.3. Mission Assurance

Mission assurance is the goal of project management and systems engineering. There are a number of ways to approach mission assurance. Aerospace Corporation has a fine description of how the company strives for mission assurance in the Fall 2007 issue of *Crosslink Magazine* [21].

Aerospace Corporation's *Mission Assurance Guide* explains six core mission assurance processes:

1. Requirements analysis and validation is a review of formally specified user requirements and their consistency with formally or informally stated user needs and expectations.
2. Design assurance is a set of planning, analysis, and inspection activities to assess whether the evolving designs can produce a system that will perform as intended overall operating conditions throughout its design life.

3. Manufacturing assurance ensures [19] that the planned manufacturing processes are repeatable and reliable and can produce the system as designed.
4. Integration, testing, and evaluation verify that assembled components meet requirements individually and as part of the finished system.
5. Operational readiness encompasses all activities required to transport, receive, accept, store, handle, deploy, configure, field-test, and operate launch and space vehicles and supporting ground systems.
6. Mission assurance reviews and audits are forms of independent technical analysis that facilitate understanding of the interfaces and composite performance of a system while synchronizing government and contractor expectations. The three types are technical reviews, audits, and readiness reviews [21].

Seven mission assurance disciplines support the six core mission assurance processes identified above in Aerospace Corporation's *Mission Assurance Guide*. These supporting disciplines “provide the more technically oriented underpinning of mission assurance application and include engineering methodologies specifically geared toward system design validation and product verification” [21]. These seven mission assurance disciplines follow:

1. Risk management is a structured approach to identifying and evaluating risk and risk-control measures and communicating mission threats to program stakeholders.
2. Reliability engineering includes the development and validation of probabilistic system reliability requirements and design trade-off studies in the early program phases. During design development and production, it assists with the determination of component failure rates and development of probabilistic reliability models, analysis of failure modes and effects, identification and control of critical items, application of worst-case and parts-stress analyses, and analysis of accelerated-life test data.
3. Configuration management seeks to control the technical hardware and software baselines of a program—the requirements, specifications, designs, interfaces, data, and supporting documentation. The goal is to ensure that the functional, allocated, developmental, test, and product baselines are consistent, accurate, and repeatable and that any changes to those baselines will be recorded and will maintain the same accuracy, consistency, and repeatability.
4. Parts, materials, and processes (PMP) engineering seeks to provide a standardized set of qualified components from which to build a reliable product at a reasonable cost and risk.
5. Quality assurance is the engineering and management discipline intended to ensure that a product meets the specified performance parameters. A well-defined and properly

implemented quality assurance program instills confidence that all quality requirements have been met through control of operations, processes, procedures, testing, and inspection.

6. System safety assurance applies engineering and management principles and techniques to control system hazards within the constraints of operational effectiveness, schedule, and cost.
7. Software assurance seeks to ensure that system software will meet performance requirements and user expectations and will be dependable, maintainable, and applicable to the user's operational environment" [21].

4. Process Flows for Developing Products

4.1. *Plan, Execute, Review, Report, and Update (PERRU)*

Every process and procedure should have a set of activities that is both consistent and thorough; each process and procedure should provide for self-evaluation. They should have a structure that can be summarized in the acronym PERRU for plan, execute, review, report, and update. Figure 1.3 illustrates this structure and data flow; it also provides some of the

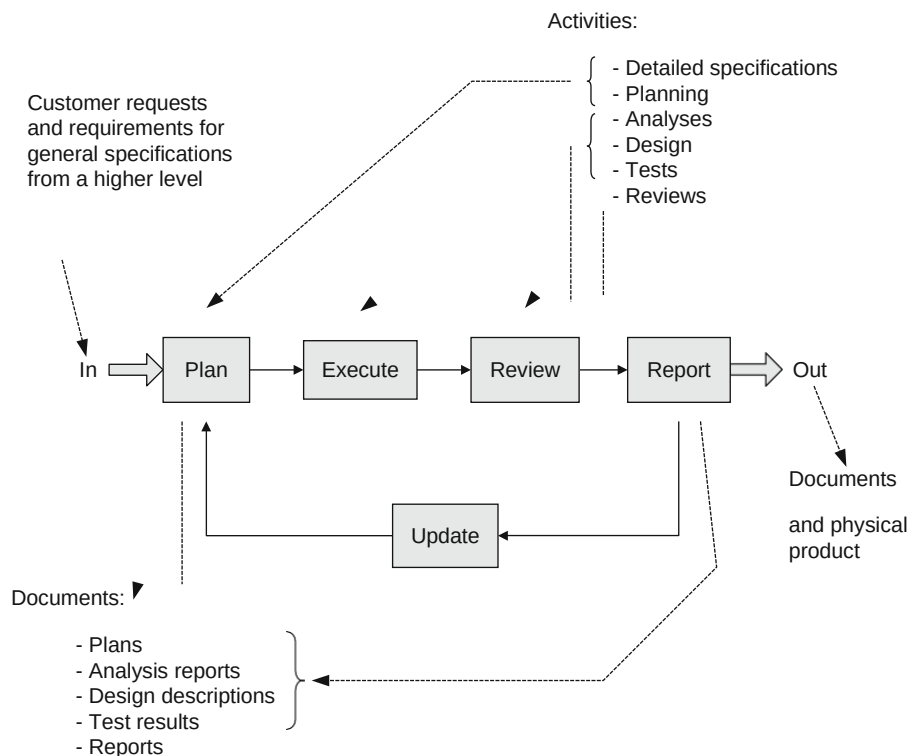


Figure 1.3: Basic, self-similar unit of operation that suits any level of detail within a project.
(© 2008–2009 by Kim Fowler. Used with permission. All rights reserved.)

types of activities and types of documents generated during a project. This structure has an interesting self-similarity principle that exists at every level of process, from high- to low-level processes and finely detailed procedures, within project and product development.

4.2. *Development Processes*

A development process provides the flow of events and procedures that should occur to develop a product. The necessary characteristics of a process follow:

- Addresses the field and market of the product
- Covers all the activities needed to complete product development
- Useful and does not force “busy work” or overburden with extraneous efforts
- Consistent with company goals
- Flexible for the constraints and demands of particular projects

4.3. *Processes vs. Procedures*

The difference between a process and a procedure is a qualitative description, as well as diversity in details. A process contains the overarching principles that drive best practices, and tends to be general. A procedure is a specific implementation of the process for a single, focused area of concern.

4.4. *General Process Models*

A process model indicates the activities and their general phasing. Process models typically comprise a visual view of those activities. The activities include concept, design, evaluation, integration, acceptance sign-off, production, and operation.

Figure 1.4 shows a simple representation of the V-model of development. The figure outlines six primary phases, and shows how verification and validation connect between various activities and across phases. (Recall that verification is an objective set of tests to confirm that the product meets the metrics of the requirements, while validation means that the product in a more subjective perspective meets the original intent.)

A limitation of the V-model is that it seems to imply that the requirements are complete in the conceptual or preliminary stage. This does not occur in most, if not all, product development. Requirements, design, and evaluation often iterate several times before final integration and acceptance. This iterative situation leads to spiral development as a

defined discipline.

www.newnespress.com

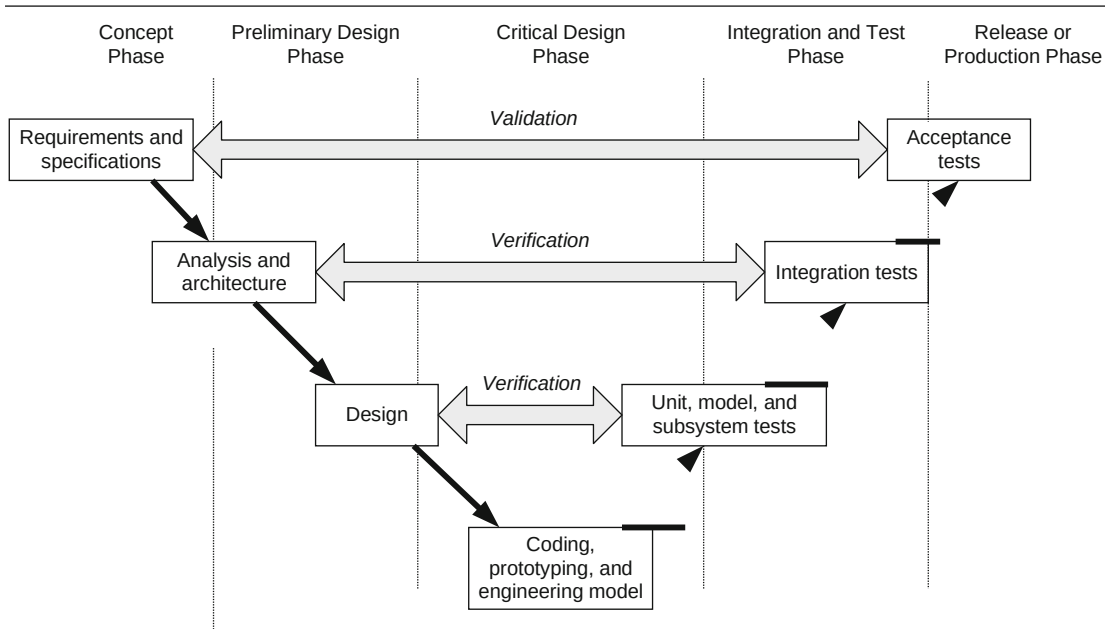


Figure 1.4: Simplified V-model of development that shows how it fits development phases.
(© 2008–2009 by Kim Fowler. Used with permission. All rights reserved.)

Figure 1.5 shows spiral development, which adds components, modules, and subsystems, in succession, to a system. Spiral development thoroughly tests the system for functionality and meeting the appropriate set of requirements after each module addition; this is one form of verification. An important aspect of spiral development is that system functionality builds in stages. Each stage plans a new set of implemented features, tests them, evaluates them, and then updates the system configuration to include them.

4.5. An Example of Phases, Processes, and Procedures

Figure 1.6 shows an example of implementing a V-model process; it requires the procedures, checklists, and all forms of documentation to be generated for a project that develops a new product. The figure displays the first level of detail in the task summaries for each phase.

Diving into the next level of detail, consider as an example the critical design phase, the phase that refines and completes the architecture and requirements for a new product. It also contributes a number of final analyses. A critical design review completes this phase.

Figure 1.7 gives the process flow for the critical design phase. Figure 1.8 illustrates an example of one potential work package (an elemental or kernel task) within the critical design phase.

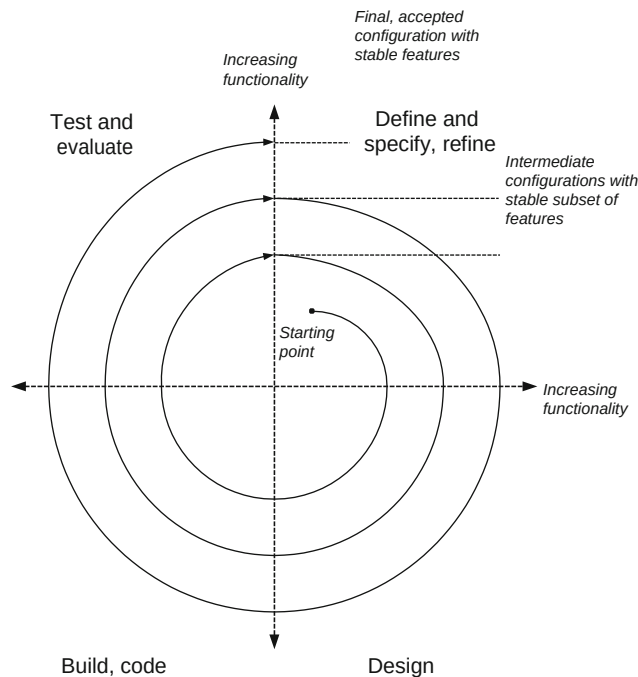


Figure 1.5: Simplified spiral model of development. A system develops in stages, and each stage passes through four separate development phases. (© 2008–2009 by Kim Fowler.

Used with permission. All rights reserved.)

The primary *responsibilities* in the critical design phase might be held by the following individuals or groups:

- Hardware (H/W) design: Electronic engineer(s)
- Software (H/W) design: Software engineer(s) or developer(s)
- Mechanical design: Mechanical engineer(s)
- Manufacturing design: Manufacturing or production engineer(s)
- Product architecture: System architect
- Requirements analysis: System engineer(s)
- Control board: Program manager, system architect, system engineer(s)

Other company employees may contribute to the architecture and requirements. All involved in the project might participate in reviews. Employees outside the project can sometimes provide unbiased considerations. Subcontractors and customers often participate in reviews.

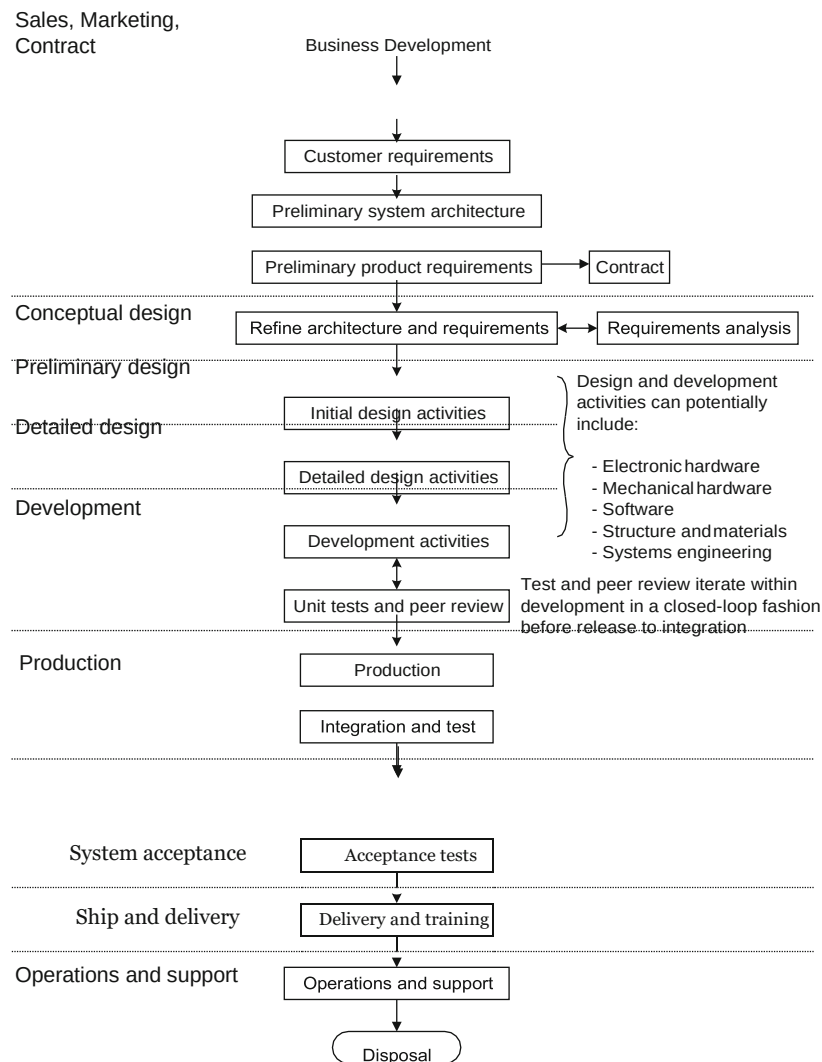


Figure 1.6: An overview of a high-level process framework and procedures for product development, production, and delivery. This is the first level of detail for task summaries. (© 2008–2009 by Kim Fowler. Used with permission. All rights reserved.)

Inputs include the requirements, the state of work (SOW) and work orders prepared by the project manager. The *outputs* include analyses (e.g., power, reliability, environmental, worst-case stress), Product Requirements Document, Interface Control Documents, development plans, manufacturing plan, test plans, and Engineering Change Notices, among many other documents.

The Critical Design Phase has unpredictable *durations* that depend on customer requirements and your company's loading with other projects. For small projects and subsystems, this phase can often take from several weeks to several months in duration. For large systems, this phase can be measured in years.

Sales, Marketing,
Contract

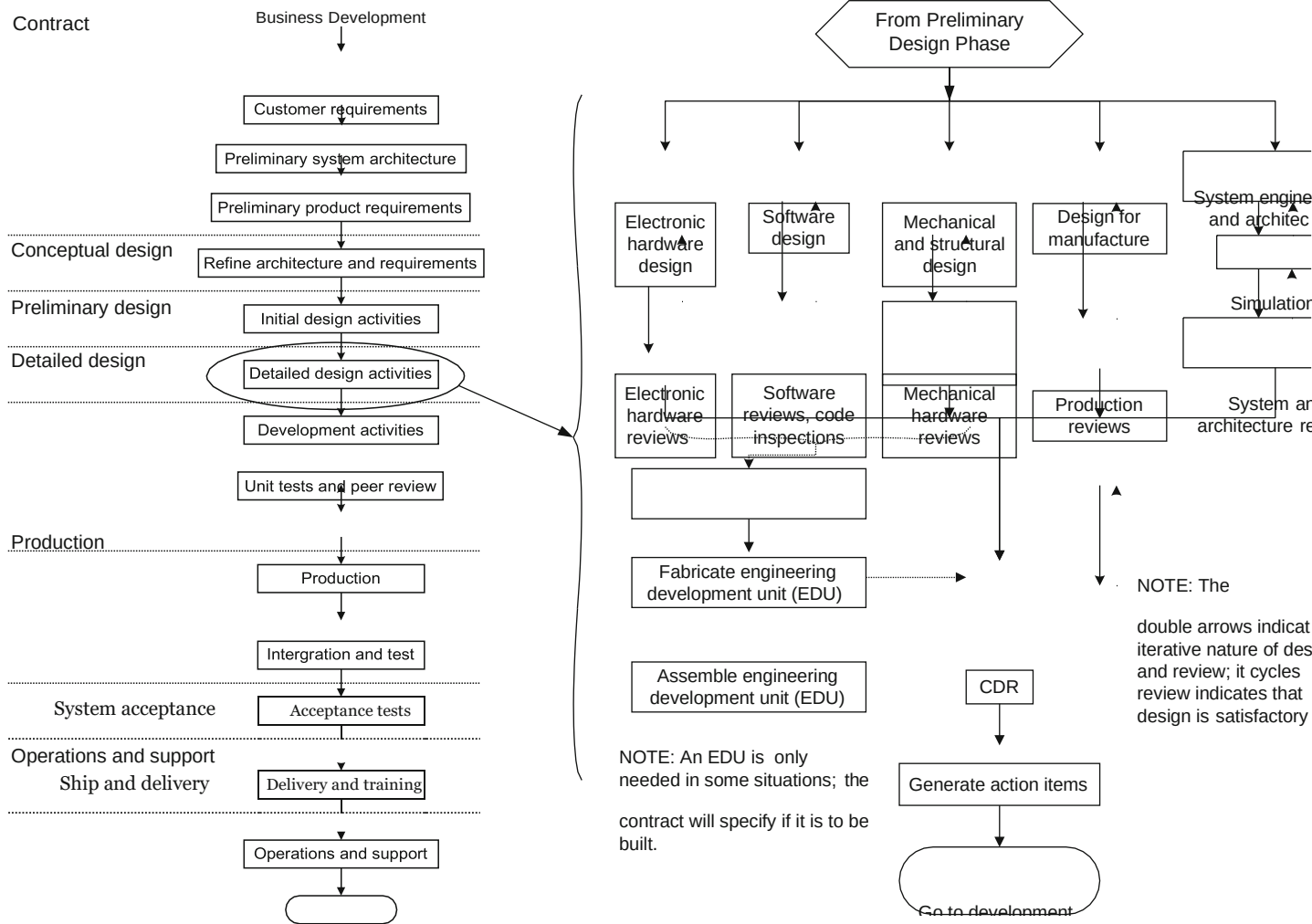


Figure 1.7: Process for the critical design phase. This is the second level of detail for task summaries. (© 2009 by Kim Fowles. Used with permission. All rights reserved.)

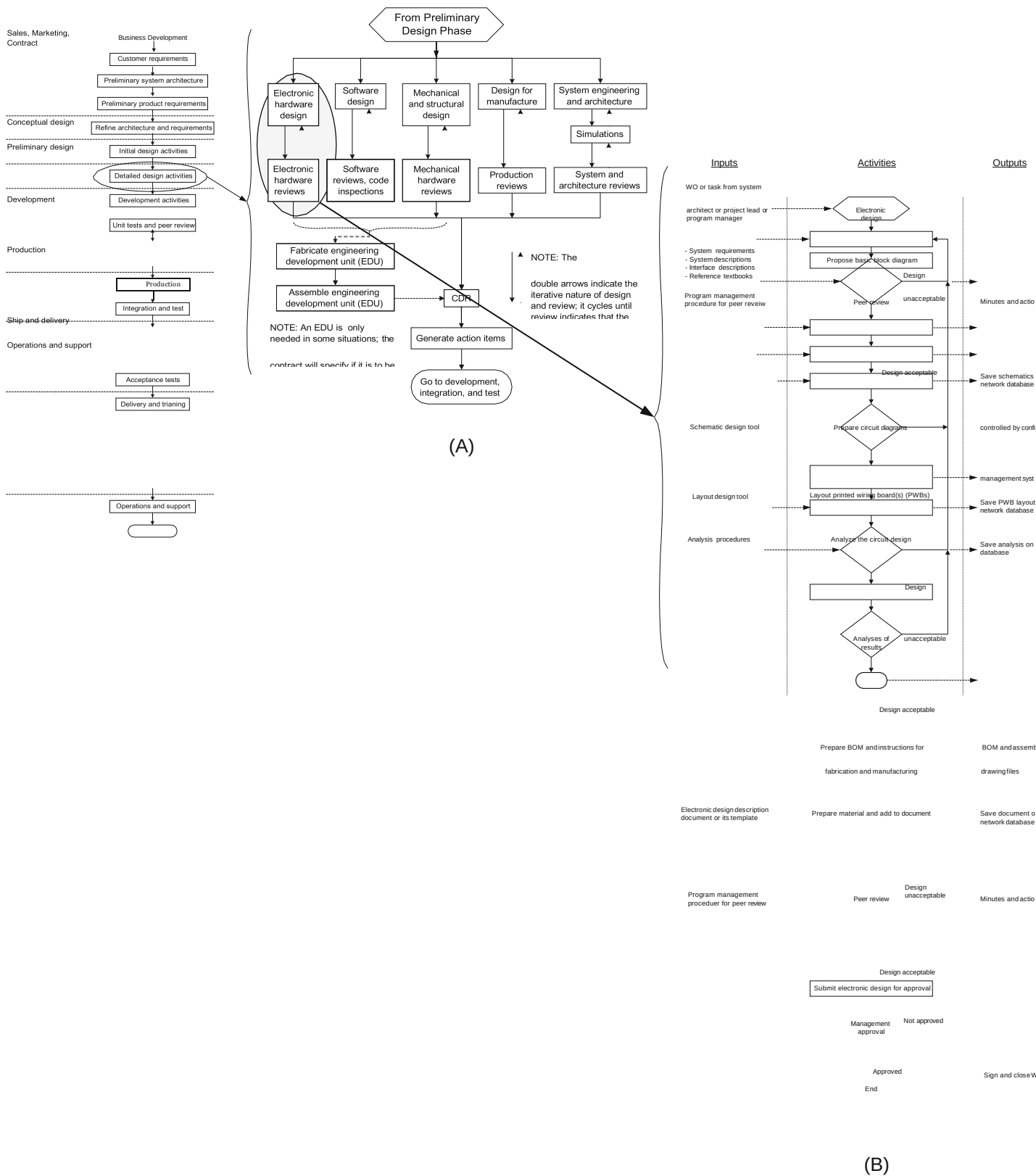


Figure 1.8: An example of a procedure for electronic circuit design during the critical design phase. This is the third level of detail and approaches a single task. (A) Location of successively more detailed levels. (B) Example of a work package or elemental, kernel task. (© 2009 by Kim Fowler. Used with permission. All rights reserved.)

5. Standards

“These lists are of necessity incomplete. There are many standards-setting organisations and standards that are specific to particular industries, types of product, national markets, etc., and changes are frequent” [22].

5.1. General Standards Organizations

- International Organization for Standardization (ISO) (<http://www.iso.ch>).
- International Electrotechnical Commission (IEC) (<http://www.iec.ch>). The electrical/ electronic section of ISO.
- U.S. National Institute of Standards and Technology (NIST) (<http://www.nist.org>). Coordinates the production of U.S. standards.
- European Committee for Standardisation (CEN). European standards are prefixed EN (Euro Norm).
- European Committee for Electrotechnical Standardization (CENELEC).
- European Commission (EC). The EC issues directives related to compliance with standards.
- British Standards Institution (BSI) (<http://www.bsi.org.uk>). The BSI coordinates the production of UK standards. British standards are prefixed BS.
- The UK Accreditation Service (UKAS) (<http://www.ukas.com>) is responsible for the accreditation of laboratories offering measurement and calibration services. UKAS also provides accreditation for ISO9000 certification organizations.
- German Association for Electronics and Information Technology (VDE) (<http://www.vde.de>). VDE produces standards for electronics and IT, provides accreditation services, and operates test laboratories. The VDE mark is affixed to equipment that complies with relevant VDE standards.

“All other industrialised countries have national standards organisations, which contribute to international standardisation work. However, only a few maintain standards that differ significantly from international or other major national standards” [22].

5.2. Industry-Based Standards Organizations

5.2.1. Mechanical and Systems

- American Society of Mechanical Engineers (ASME) (<http://www.asme.com>).
- American Society of Testing and Materials (ASTM) (<http://www.astm.com>).

5.2.2. *Electrical and Electronic*

- Institute of Electrical and Electronic Engineers (IEEE) (<http://www.ieee.org>). (The IEEE has 37 societies that generate standards and organize seminars on a variety of aspects of system development.)
- Telcordia Technologies (was Bellcore). Telcordia standards are applied in the telecommunications industries worldwide (<http://www.telcordia.com>).

5.3. *Military Standards Organizations*

- U.S. Department of Defense—Military standards, handbooks, and specifications (MIL-STDs, MIL-HDBKs, MIL-SPECs).
- North Atlantic Treaty Organization—NATO Standards (STANAGS).
- UK Ministry of Defence—Defence Standards (DEFSTANs).

5.4. *Aviation and Aerospace Standards Organizations*

- International Civil Aviation Organization (ICAO).
- Air Radio Inc. (ARINC). Produces standards for aviation electronic systems (avionics).
- National Aeronautics and Space Administration (NASA).
- Federal Aviation Administration (FAA) (USA).
- Civil Aviation Administration (CAA) (UK).
- European Space Agency (ESA).

6. Potential Procedures, Checklists, and Documents

A company building mission-assured, mission-critical, or safety-critical systems has six primary areas that contribute to the operation of the business. These areas are:

- Business
- Marketing and sales
- Personnel

- Infrastructure
- Manufacturing and production
- Engineering

Each area has a number of procedures that are to be followed within its purview.

Business deals with company organization, governance, relationships (e.g., suppliers, bankers, tax accountants), legal concerns, government regulations, facilities, and office equipment, among other things. Marketing and sales deal with initial contact and customer relations, order fulfillment, billing, and service support. Personnel and human resources deal with hiring, benefits, training and development, personnel files, and termination. Infrastructure has to do with equipment and systems and facilities to carry out the company business; it is often split among several groups. Manufacturing and production deals with inventory, fabrication, assembly, and distribution; sometimes it provides service support as well.

In this book, we are only focusing on the development engineering aspects. Following is a partial list of major concerns for which engineering has responsibility. (Clearly other disciplines exist; this listing gives an indication of the types of issues.)

1. Project flowcharts
2. Project tracking—Engineering concerns
 - Visibility
 - Accountability
 - Configuration management
 - Version control
 - Database implementation
 - Engineering change notice
 - Feedback
 - Improvement
 - Closeout
 - Archiving
 - Outsourcing
 - Qualification of vendors
 - Templates—Plans, documents, reports, action item, design documents, test results

3. General engineering processes

- Requirements
 - Input—Customer, company,
 - standards Analysis
 - Producing
 - metrics
 - Modification
 - and change
 - Compliance
 - matrix
- Analysis—Business feasibility
- Design
- Reviews
 - Peer review
 - Control board review
 - Design review—CoDR, PDR, CDR, PRR
- Development
- Test
- Integration
- Acceptance
- Delivery

4. Electronic and electrical engineering

- Tools
- Tool certification
- Design
 - Schemati
 - cs
 - Circuit board
 - layout Assembly
- Analysis
 - Signal integrity

Electromagnetic
compatibility (EMC)

Power consumption

Ther
mal

Radia
tion

Shock and
vibration

Humidity

Corrosion

Worst-case stress

5. Software engineering

- Tools
- Tool certification
- Style guide
- Source listing
- Compilation and transfer to hardware

6. Mechanical engineering

- Tools
- Tool certification
- Design

Schemati
cs

Mechanical

layout

Assembly

- Analysis

Mass, size, and
volume Materials

Structural
model

Thermal

Radiation
Shock and
vibration
Humidity
Corrosion
Worst-case stress

7. Systems engineering

- Tools
- Tool certification
- Design
 - Schematics
 - Mechanical layout
- Analysis
 - Event tree analysis (ETA)
 - Failure mode effects analysis (FMEA)
 - Fault tree analysis (FTA)
 - Reliability
 - Radiation
 - Interface inspections
 - Integration coverage—functional, physical, environmental

8. Metrology

- Qualification of vendors
- Calibration
- Certification of measurements
- Archiving

9. Service support

7. Review of Procedures and Processes

7.1. *Difference between Procedures and Processes*

The difference between processes and procedures is a qualitative description, as well as a difference in the detail. A *process* contains the overarching principles that drive best practices. It tends to be general. *Procedures* are specific implementations of the process for a single focused area of concern; often they are step-by-step instructions.

7.2. *Why Review Procedures and Processes?*

Procedures and processes characterize every system of business or technical development. These systems are necessarily limited due to flaws, gaps, or changing circumstances. Nevertheless, a system should have an avenue for audit, self-examination, correction, change, and update.

Reviews of procedures and processes are a quality assurance function. These reviews may be directed by either external or internal sources. Customers, audit agencies, and consultants may conduct external reviews with the agreement of your company. Employees of your company and consultants might conduct internal reviews.

7.3. *Types of Review*

Design and Peer Review: Effective review of the project should have a stereotyped format including agenda, checklists, and minutes. An outline example of review minutes follows:

- Date of the review
- Review agenda
- Who attended
- Who presented the design
- Lead and independent reviewers
- What major decisions were made
- What action items were generated, their due dates, and who is responsible for each.

Reviews should generate action items to ensure that identified issues are addressed. All action items should be tracked in a database. Each action item should have the following fields:

- Unique identifier number
- Status (open, closed, in work, in sign-off)

- Date opened
- Brief summary
- Response summary
- Requestor
- Assignee
- Due date

Design Reviews: For larger programs and projects, the design review should have independent reviewers who are not directly associated with the project. The review committee for each formal review should consist of at least four members plus a designated chairperson; none of which should be members of the project team. I have to say this rarely happens—it is the ideal but most companies do not have the people, time, or resources to devote to independent review.

Most companies organize design reviews with members of the project team presenting and reviewing. Sometimes they will ask a customer or client to attend and critique. This form of review is still effective.

Regardless of the format, you should send a review package to each reviewer about

two weeks ahead of the review. The review package should contain a copy of all the slides to be presented at the review along with appropriate background material.

Code Inspections and System Reviews: Software code walkthroughs are a legitimate

form of review as well. They are a form of static testing, but highly effective. These forms of peer review are an excellent way to encourage proper designs and good development processes. For some types of products, such as medical devices, they are an important part of the formal development. You should still have procedures, which can be simple and straightforward, for recording notes or minutes and then maintain a database of action items.

This same sort of review would be good for early system integration. If done regularly by team members, these reviews would reveal more problems earlier and give you a better chance at ironing them out sooner.

7.4. *Frequency of Review*

External reviews of procedures and processes for quality assurance should occur at least once a year. Your company might have a frequency goal of two or more external reviews each year—but reviews do take time.

Internal reviews of procedures and processes for quality assurance should occur at least twice a year. Your company might have a frequency goal of four internal reviews per year.

7.5. *Review Content*

Reviews of procedures and processes for quality assurance should address the following questions:

- What processes and procedures are being followed?
- What processes and procedures are not being followed?
- What processes and procedures are not sufficient?
- What processes and procedures need updating?
- What processes and procedures need revision?

7.6. *Course of Action, Changes, and Updates Following Review*

Reviews of procedures and processes for quality assurance will proceed according to the following general course of action:

- Agreement to conduct an audit within your company or between your company and an outside party.
- Schedule the audit.
- Conduct the audit.
- Review the findings. In general, answer the questions immediately above.
- Perform a general review of the procedures and processes.
- Combine the findings of the audit and general review and present to your company's control board.
- Implement adjustments, corrections, and updates to your company's procedures and processes.

7.7. *Review Responsibilities*

Customers, audit agencies, and consultants may conduct external reviews. Employees of your company and consultants may conduct internal reviews. Management determines when the reviews and audits may take place. Your company's control board is responsible for implementing the revisions or updates that derive from a review.

8. Configuration Management

8.1. *Rationale for Configuration Management*

Every business or technical development system needs configuration management because problems, mistakes, and changing circumstances occur frequently, if not continuously. Configuration management helps reduce the consequences of these problems from becoming too disruptive by maintaining an understanding of current system state.

8.2. *Configuration Management Coverage*

Configuration management monitors the state of just about everything in a project. Here are some examples of areas within configuration management:

- Software
- Hardware
- Products
- Identification
- Database and network

8.3. *Records Responsibility*

Your company's control board is responsible for implementing configuration management. The project manager then may designate who will manage the configuration of a particular project.

8.4. *System and Location*

Maintain the configuration management system in a database on a server located on the premises. The data on the server should be backed up regularly to an off-site storage system. Back up the remote site once a day if possible but once a week at the very longest. Should the system fail or be put out of commission, the data can be retrieved and restored on a replacement system.

8.5. *Version Control*

A centerpiece of that configuration management is the version control system. Version control is software that resides on the system server and archives project data so that it may be traced from inception to the end of the project. A number of commercial software packages implement reasonable version control systems.

Version control has four different functions:

- Performs the library operations of checking files and documents in and out of the system in an unambiguous fashion. Prevents two or more people checking in the same file, each with different modifications, and having one version overwrite and destroy the other modifications. Can merge modifications so that all changes are recorded.
- Uniquely assigns identification to each file so that it can be traced, tracked, followed, and found.
- Records all changes to each file and the list of versions of each file. This allows someone to “unroll” the changes and return to a previous version, if so desired.
- Time-tags and date-tags every file and each revision to any file.

8.6. *Design Repository*

One particular portion of a project’s configuration management is the design repository. This repository contains the design descriptions and instructions for production. It is a particular “folder” or “file drawer” in the project’s storage area that contains these files of the design descriptions and instructions. Version control, as mentioned, controls document storage into and retrieval from the design repository.

8.7. *File Structure*

Each project has its own separate storage area. Here is an example of partitioning, in which the files can segregate into the following areas, but these are at the discretion of the program manager or project lead:

- Design repository
 - Architecture—Requirements, descriptions
 - Software—Source listings, requirements, descriptions
 - Hardware—Schematics, requirements, descriptions
 - Sensors and optics—Schematics, requirements, descriptions
- Business operations
 - POs
 - Work orders (WOs)

Contract, request for proposal (RFP), rough-order-of-magnitude (ROM) quotes, SOW

- Products

Manufacturing and
fabrication instructions In-
process routers (IPRs) for
production Subcontractor
reports
Identification

- Documents

Plans
Descriptio
ns Test
results
Checklists

- Records

Minutes—Peer reviews, design reviews (CoDR, PDR,
CDR, MRR, TRR, PSRR), control board meetings,
failure review board meetings

Logs—Problem reports/corrective actions
(PRCAs), engineering change requests (ECRs),
engineering change notices (ECNs)

Test results—Bench tests, unit tests, integration,
acceptance tests Communications—Memos, email
messages, notes, letters, customer audits

8.8. *Obsolete Documents*

When documents or company procedures or processes change, revise, or become subsumed, the original must be removed from the active templates folder. They should be placed in a folder or location marked “obsolete.” Each obsolete document, procedure, and process should have a watermark of “obsolete” inserted on each page when they are removed from circulation.

Another concern is for paper copy of documents being obsolete when compared to the current version in the design repository. A good practice to put a disclaimer in the header or footer of each page to the effect that “This paper copy is for reference only—please consult the company database repository for the current version.”

8.9. *Training for Use of the System*

A quality assurance team or management-designated personnel should train staff in the use of the configuration management system. Training should cover the basic operations, some of which are listed in the following:

- Overview of the configuration system and version control
- Entering the system
 - Where to find it on the server database
 - User ID and password
 - Navigation through system
- File transfers
 - Entering a new file into the system
 - Retrieving a file from the system
 - Returning a file to the system
- Reviewing file archives
- How the system is set up and run
- Backups
- Restoration

9. Documentation

9.1. *Rationale for Documentation*

Documentation generally serves three purposes:

- Record the specifics of development (the “who, how, and why”) that include, but are not limited to, engineering notebooks, software source listings, schematics, and test reports. These records help when modifications, upgrades, fixes, and recalls occur.
- Account for progress (the “what, when, and where”) toward satisfying requirements and provides an audit trail of the development. These types of documents include memos, but are not limited to, meeting notes and minutes, review action items, and project plans. An appropriate plan for documentation supports rigorous testing, validation, and verification.

- Instruct users and owners on the extent of functionality of your product. Instructions include, but are not limited to, user manuals, DVDs with instructions, websites, and labels and warnings that present concise instructions [23].

9.2. *Coverage and Responsibility for Documentation*

Documentation addresses every aspect of business; it explains the “who, what, when, where, why, and how” of a project. For your company to develop a product, there are two primary types: company documentation and project documentation. The appendices at the end of this chapter provide examples of both for a smaller project, such as a self-contained piece of equipment (not a large structure or vehicle such as an aircraft, which is a domain unto itself).

The project manager designates who will manage the documentation and the document plan. Typically everyone on a project team contributes specific documents that represent their area of work.

9.3. *Types of Documentation*

Many types of documents exist. Each set is project-type specific. The appendices at the end of this chapter provide examples of types of documents for a smaller project. [Figure 1.9](#) illustrates several of the major documents that direct development for a smaller project.

The set of documents for a project should answer the “who, what, when, where, why, and how” of product development. Here is a general outline of the types of documents that are part of most products:

- Plans (who, what, where, when, why)
- Design documents (how, why)—Source listings, schematics, engineering notebooks
- Reviews and reports (who, what, where, when, why)
- Instructions (what)—User manuals, brochures, training materials, maintenance and repair guides

The appendices at the end of this chapter provide example outlines of some documents for a smaller project.

9.4. *Best Practices for Documentation*

First, understand the audience for each document and then tailor the document to that audience. That means an appropriate level of detail and reading comprehension with an

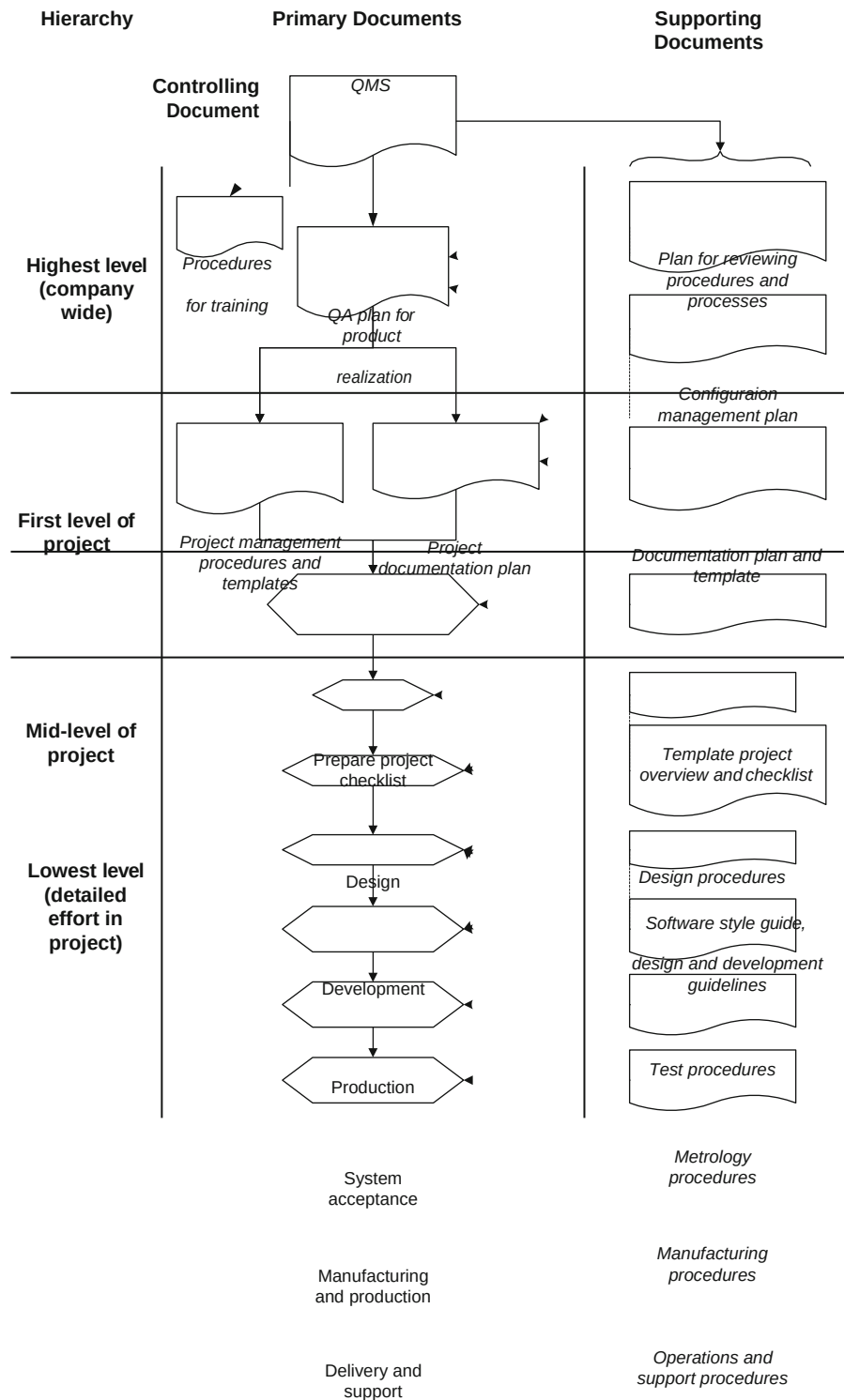


Figure 1.9: Example of high-level document tree for direct development of a smaller project, such as a self-contained piece of equipment; not recommended for a large structure or vehicle such as an aircraft. (© 2009 by Kim Fowler. Used with permission. All rights reserved.)

intuitive format or instructional flow in the layout of text and graphics. Every document should have attributes in the following order of priority:

1. Correct
2. Complete
3. Consistent
4. Clear
5. Concise

Good documents should share similar characteristics that cover the following points:

- Basic content

What: Prominently note the model number and version
Why: Clearly explain the need or theory
How: Detail the operation

- Clear organization

Introduction: Purpose, Scope, Definitions, Acronyms, and Abbreviations, References
Overview, with rationale
Description, report, or procedures
Appendices

- Modular format
- Clearly illustrated figures and tables
- Detailed schematics
- References to source listings and other technical records
- Table of contents, index, and cross-references

Three heuristics about documentation that I have learned follow:

- Documentation is integral to every product.
- While good documentation cannot help a poor product, poor and inadequate documentation can destroy a good product.
- While some good products have poor documentation, I have *never* found a poor product with good documentation [23].

www.newnespress.com

References

- [1] Blanchard BS, Fabrycky WJ. Systems engineering and analysis. 4th ed. Upper Saddle River, NJ: Pearson Prentice; 2006.
- [2] International Council on Systems Engineering. A consensus of the INCOSE Fellows. Available at www.incose.org/practice/fellowscensus.aspx.
- [3] Maier MW, Rechtin E. The art of systems architecting. 2nd ed. Boca Raton, FL: CRC; 2002.
- [4] Mission assurance. Available at: en.wikipedia.org/wiki/Mission_assurance; 2009.
- [5] Mission assurance—government needs to manage risk differently in face of new operating realities. Booz Allen Hamilton Inc.; March 8. Available at: www.boozallen.com/consulting-services/services_article/659316; 2004.
- [6] Gordon P. Ground zero for mission assurance, PRTM. Available at: www.prtm.com/uploadedFiles/Strategic_Viewpoint/Articles/Article_Content/PRTM_Ground_Zero.pdf; 2006.
- [7] Mission critical. Available at: http://en.wikipedia.org/wiki/Mission_critical; 2009.
- [8] Farlex. The free dictionary. Safety-critical system. Available at: encyclopedia2.thefreedictionary.com/safety-criticalsystem; 2009.
- [9] Sedlak JR. Quality management system aerospace requirements: differences between ISO 9001:2000 and AS9100B, Akron, OH: Smithers Quality Assessments, Inc; August. Available at: www.isots16949.com/AS9100B-Differences-from-ISO9001.pdf; 2006.
- [10] AS-9100 Standard—QMS. Available at: www.siliconfareast.com/as9100.htm.
- [11] Penn L. CMMI and Six Sigma. In: Chrissis MB, Konrad M, Shrum S, editors. CMMI: guidelines for process integration and product improvement. 2nd ed. Addison-Wesley: Upper Saddle River, NJ; 2007, pp. 5–8.
- [12] Watts H. CMMI: history and direction. In: Chrissis MB, Konrad M, Shrum S, editors. CMMI: guidelines for process integration and product improvement. 2nd ed. Addison-Wesley: Upper Saddle River, NJ; 2007, pp. 5–8.
- [13] Verzuh E. The fast forward MBA in project management. 2nd ed. New York; 2005, p. 3, 7, 8, 20, 21, 113, 114.
- [14] MIL-STD-499B (draft). Systems engineering; May 6, 1994.
- [15] Oshana R. DSP software development techniques for embedded and real-time systems. Burlington, MA; 2006, p. 507–509.
- [16] Reinertsen D. In search of the perfect product specification. IEEE Instrumentation Measurement Mag 2000;3(2):28–31.
- [17] Fowler K. What every engineer should know about developing real-time embedded products. Boca Raton, FL: CRC; 2008, p. 32.
- [18] Pazemenas V. Rapid development for medical products. IEEE Instrumentation Measurement Mag 2000;3(2):32–37.
- [19] Test-driven development. Available at: en.wikipedia.org/wiki/Test-driven_development; 2009.
- [20] Grenning J. Applying test driven development to embedded software. IEEE Instrumentation Measurement Mag 2007 (December):21.
- [21] Guarro S. The mission assurance guide: system validation and verification to achieve mission success, Crosslink Mag 2007;8(2). Available at: www.aero.org/publications/crosslink/fall2007/03.html.
- [22] O'Connor P. Test engineering: a concise guide to cost-effective design, development and manufacture. New York: Wiley; [Reliability, testing and related regulations and standards. Available at: www.pat-oconnor.co.uk/reteststds.htm]; 2001.
- [23] Fowler KR. Electronic product development: architecting for the life cycle. 2nd ed. New York: Oxford University Press; [forthcoming].

CHAPTER 2

Failsafe Software Design: Embedded Programming in a Fail-Certain World

Jeffrey M. Sieracki

1. Software Matters

Ask yourself why you trust your microwave oven. Think about that as your daughter stares in the window and watches her milk warm. There in your kitchen is a ubiquitous lifestyle tool with components that could, in seconds, do severe harm to you or your family. Yet careful design of the physical cavity, the door, and various electronic and software interlocks render it as benign as a cereal spoon in most circumstances and far safer than the ordinary kitchen range next to it.

Meanwhile, tired and grumpy in the morning, you push a few buttons trusting utterly that your tea water or frozen sausage will heat up to your liking. A few beeps, the numbers glow, the magnetron energizes, and breakfast is served. The buttons always respond, the timer always counts down to zero, the door interlock always knows when you are in a hurry and safely disengages the innards. Somebody cares.

In my kitchen, on the wall across the room, hangs another piece of modern machinery:

A two-line telephone by one of those late 1990 companies formerly known as AT&T. It's lovely to look at, includes autodialing, conferencing, speaker phone, and two-line caller ID. The large LCD display is bordered by four "soft" buttons for menu control and—herein lies the rub—a red "new call" LED for each of the two lines.

As calls come in, the caller ID system dutifully decodes the name and number of the caller, recording this information together with the time, date, and incoming line, and the embedded software lights the "new call" LED for the appropriate line. Each call is marked "new" on the text screen until I review it once by sequencing through the list.

Curiously, sometimes as I sequence back through the list until I have seen the last new call, the LED steadfastly remains illuminated. If I sequence back through again, I see the software has clearly removed the "new" designation from each listed call; yet even if I delete the calls one by one, erasing the entire caller-ID list, the LED cheerfully continues to

announce “new call.” If I pull the power to the phone, its clever brain keeps track of the situation and my Hal 9000 wall phone instantly boots back up glowing as if to announce, “I’m sorry Dave, but I can’t allow you to turn off that LED.” In designing this phone, somebody didn’t care quite so much.

It is possible to turn off the LED by a special sequence of strokes that clears memory, losing all record of all calls. In other words, the “soft reset” works. It didn’t take me long to think of that. In fact, the first time I witnessed my persistent “new call” light, I recognized the failure mode as an old friend and began tinkering to see which software pathways had been addressed and which had not. I have not only seen such an error before, I’ve created one. The circumstances were far less benign.

This chapter is about good software design for mission- and safety-critical systems. We will review simple principles and practices that help ensure systems work well and keep working. Our guiding mantra is *Assume Fallibility*.

We are generous in this assumption; don’t just blame your sales team or the sketchy documentation from your system architect. Assume fallibility in the hardware you are controlling, in the sensors you rely upon, in the next programmer that picks up your code, in your communications channels, in the idiot-proof end users, and, ego to the wind, in yourself. We will draw frequently from real-world examples to reinforce these points.

This chapter is *not* about the myriad guidance and regulating documents available for safety-sensitive and mission-critical software design. Certainly to those working in specific fields, documents such as DO-178b (avionics), MIL-STD-498 and -882 (military), and 21 CFR 820 (medical devices), will eventually be of keen interest. We shall instead consider some key unifying principles behind such guidance documents and attempt to motivate further interest in what sometimes appears to be tedious recitations of burdensome process.

Modern software engineers are often heavily schooled in how to employ good process and design strategies to find and avoid bugs. But in contrast to more physical engineering fields—electrical, mechanical, or civil—there is little emphasis placed on designing and compensating for failure modes. Where other fields employ redundancy and experience-driven safety factors as a matter of course, software developers are often tying their own hands in a tradition of minimalism and false assumptions.

This chapter is laid out in sections, each building on the previous. [Section 2](#) is about process, and how and why it can work to your favor in engineering reliable critical systems.

[Section 3](#) delves into important core principles and patterns that arise repeatedly at all levels of failsafe design and implementation. [Section 4](#) discusses the user interface and the key role it may play in the success of a mission—or safety—critical system. [Section 5](#) addresses some of the design patterns in the context of the common practice of creating one's own real-time operating software. Finally, [section 6](#) considers an extended “what-if” scenario in which we examine a failsafe hardware design and consider how it might be usefully mimicked in software.

www.newnespress.com

The recommendations and principles presented here were developed over many long years of both book learning and hands-on, hard-knocks experience. The topics will be of interest to applications software engineers moving into mission-critical/safety-sensitive systems work, as well as to hardware engineers transitioning to software work. They may also be of interest to those already in the field, either looking for new ways to think about their work or looking for material to help their manager understand why solid safety- and mission-critical software doesn't often come cheap.

2. The Essence of Process

The development of engineered systems, and of software in particular, has been characterized from the start by evolving descriptions of *process*. The classic waterfall process, first defined in the 1950s, lays out a feed-forward effort in which requirements feed into design, design to implementation, implementation into test and verification, and so on down to delivery. The heart of the idea, sometimes referred to as the *unmodified* waterfall model, is illustrated in [Fig. 2.1\(A\)](#). Why “unmodified”? Because in practice, the waterfall model is nearly always modified.

The waterfall model is just an attempt to sketch on paper how engineers interact in sequential stages to produce a product. Many practitioners would add pre-process steps, such as a *concept* phase, and post-process steps, such as *maintenance*, as shown in [Fig. 2.1\(B\)](#). Many would further break down the stages into sub-blocks: *design* might become *analysis*, *design*, and *test design*, while *implementation* might become *module implementation*, *module test*, and *integration*. One example of a more refined form of the waterfall is shown in [Fig. 2.1\(C\)](#).

The added degrees of detail are attempts to capture a very complicated reality. Any engineer with real project experience could sit all day adding new boxes to the diagram—and probably remain unsatisfied. In fact, a true waterfall process is almost never used.

The exceptions are those rare cases where there is no time for, no money for, or simply no possibility of changing one's mind. A one-off product, for example, might be sent out the door, ready or not, on an absolute limited budget; likewise, a satellite in orbit is not easy to reengineer once the final test data (launch!) are recorded.

Yet, even in those less common cases, there will almost certainly be some feedback between development stages—a prototype implemented gives information that changes the design, a test failed that changes the implementation—these small elements of feedback occur all the time. Even the

termination of the process in “maintenance” is artificial since, nearly always, the results of any given project will feed into the next-generation product.

The closed-loop nature of most actual development leads to the spiral model of the development process, which simply recognizes the natural cycling between *requirements definition, analysis, development, and testing*. Various authors when describing the spiral call

www.newnespress.com

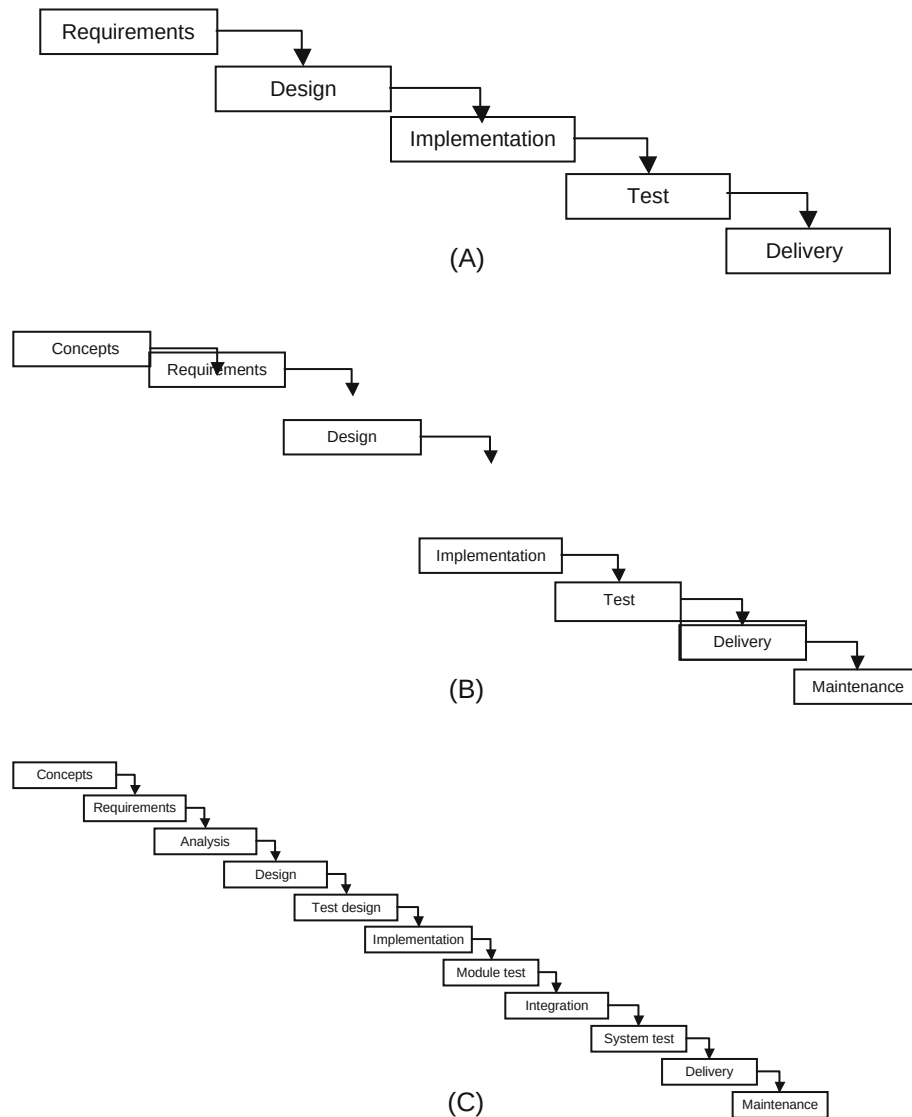


Figure 2.1: (A, B, C) The waterfall process diagram is a good metaphor for thinking about development, but does not capture the complicated reality of most engineering projects.

out different elements of the process, but the important concept is that good engineering is very seldom linear. Rather, good engineering is usually an iterative process of refining concepts and designs to converge on a solid result.

Attempts to better codify the development process have led to a host of ideas, and libraries of books on the topic. In addition to learning more about the waterfall and spiral methods, the avid reader may enjoy perusing the Internet for keywords like “agile,” “extreme,” “iterative,” “RUP,” and “RAD,” and seeing where those links take them. Attempts to firmly and definitively separate one process model from another or to quantify their respective failings fill nearly as many volumes as descriptions of the methods themselves.

At heart, the *process* chosen is often much less critical to success than the simple existence of a process. In fact, most guidance documents come to a similar conclusion. Well-known quality certifications, such as ISO 9001, do not specify a particular process; rather, they specify the existence of a process and consistent supporting documentation. The development process itself remains up to the organization. ISO 9000 is a body of quality standards, not an engineering guideline, but for purposes of applying process to mission- and safety-critical systems, thinking in terms of quality management is not far off the mark.

The purpose of defining a development process from a critical system perspective is to provide a clear framework for *provability* of the system. In the next section, provability is discussed as one of the fundamental principles of failsafe software design. In essence, the goal is to demonstrate that the system will perform as intended.

We typically speak about meeting requirements and specifications in terms of *verification* and *validation*. In general usage, *verification* refers to the process of ensuring that the project elements meet specifications, while *validation* considers whether the project satisfies its intended purpose. The common use of the two together, typically shortened to V & V, considers that the line between one and the other is sometimes fuzzy.

With this in mind, let us consider one more process model that captures concepts in a different way: the *V-model*, illustrated in [Fig. 2.2\(A\)](#). As drawn, the process clearly tends to flow from left to right, beginning in project definition, traversing implementation, integration, and test, and ending in operations and maintenance. However, the time arrow is only intended to indicate general flow, since information gained during test may easily feed back to a new iteration of coding or design. The V-model is not so concerned with a firm sequence of steps; one may use spiral, iterative, waterfall, or any other sequencing ideas to move the development team from requirements to delivery.

Rather than sequencing, the V-model is largely focusing on testability. The most important links of the diagram, from our perspective, are the *verification-and-validation* (V&V) flow arrows bridging the gap of the V. Definitions including requirements and specifications, analysis and architecture, and detailed design are set at stages on the descending leg on the left and are matched to corresponding tests, including system tests, integration testing, and module test at each ascending stage on the right.

System tests will naturally include testing of failure modes. We will see in the course of this chapter that encapsulating

mission- and safety-critical elements of the design in ways that they can be isolated, implemented, and tested independently, goes a long way toward provability of those elements. Encapsulation lies at the heart of the *design-for-test* concept as it applies to software.

Iterative, stepwise, documented testing allows the engineer to build the critical elements of the system on a rock solid foundation. In particular, risk assessment and hazard analysis

www.newnespress.com

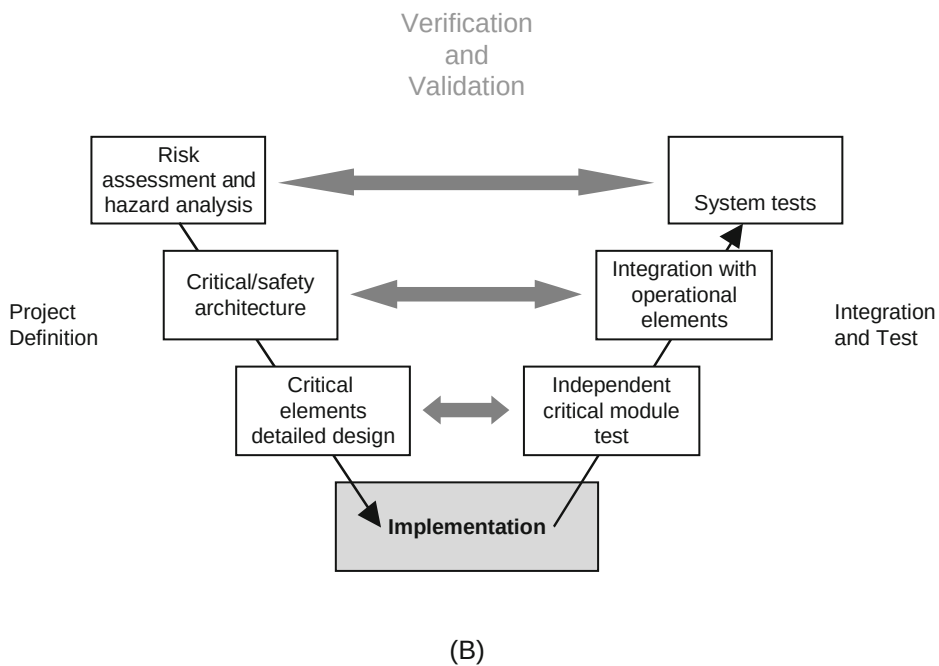
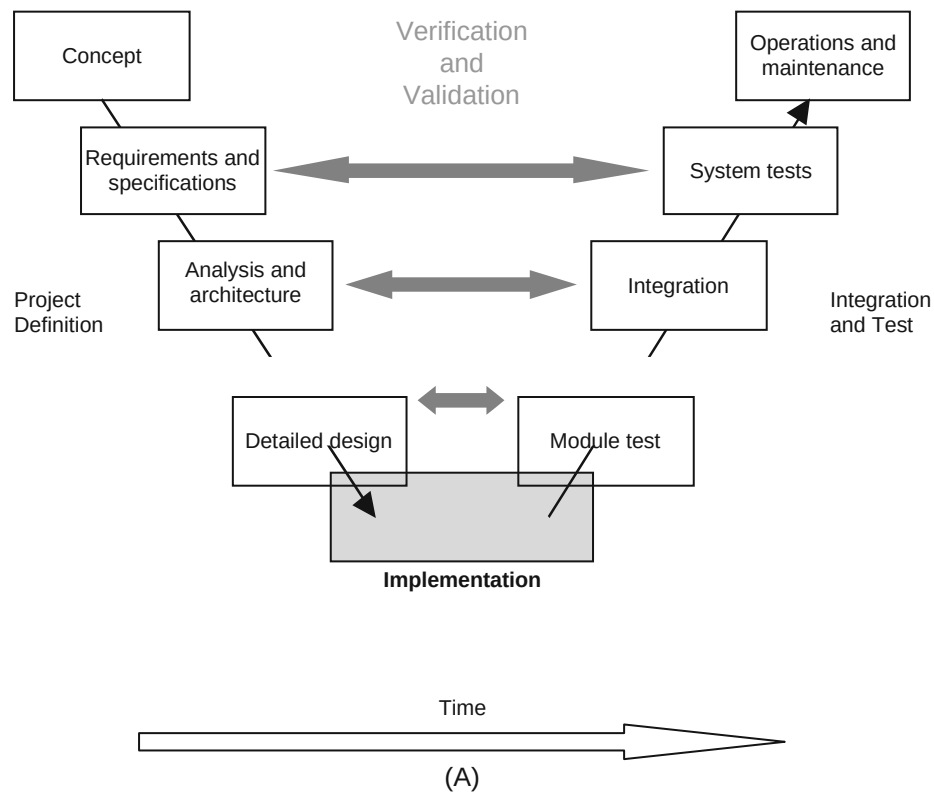


Figure 2.2: (A, B) The V-model of process reflects the ties between design and test aspects of a project. It can be adapted as a very good metaphor for understanding the process of developing, testing, and integrating safety- and mission-critical project pieces.

(RA/HA) become integral elements in the requirements and, subsequently, in the testing. [Figure 2.2\(B\)](#) illustrates how mission- and safety-critical elements can, and should, be called out, independently implemented, and independently verified. This critical-element subset of the V-model neatly captures how process relates to failsafe software design.

Any process model of software development is, at best, an approximation of reality, and the V-model is no better in that respect. The extreme emphasis on testing certainly will not appeal to applications programmers driven by tight deadlines rather than by hard mission- or safety-critical requirements. Rigid adherence will often be overkill for less demanding aspects of a project, such as general operations software. However, the V-model is an excellent graphic guide to best practices of traceability and provability for critical systems.

3. Three Principles for Design and Coding

3.1. *What Does It Mean to Be Failsafe?*

Having emphasized the importance of V&V in critical-system development, let's take a step back and remember that testing isn't everything. One problem with overemphasizing the design and test philosophy and relying on a third-party verification process is that if we design our tests to only test what we designed, then we are potentially blinding ourselves to unforeseen interaction of system elements.

Understanding what it means to be failsafe means understanding what can fail. We need

to allow for unexpected failures both inside and outside the box. In a tip of the hat to Gödel's classic logic proof, we need to recognize that we are nearly certain to experience failure modes in the field that were not realistically possible to anticipate in our design reviews.

Defining your embedded system as "the box," consider first the outside-the-box elements. As the saying goes, nothing is idiot proof, given a sufficiently talented idiot. Your users are probably not idiots, in most cases, but they don't necessarily think like you do. It is almost a certainty that your use cases and failure charts will not capture all the possible variations of input the system will ever see. Nor can you completely control the environments in which your system will be used—this is particularly true when the system has a long life cycle, such as medical or military projects that are likely to persist as legacy systems long after your primary design life.

Example 2.1. Outside-the-Box Testing

Sometimes what is obvious to the designer is not so obvious to the user. A medical test system required us to design a graphical interface that allowed patients to indicate their pain or other sensations by drawing on diagrams of their body. We very carefully thought out the

(Continues)

www.newnespress.com

Example 2.1. Cont'd

interaction: The user circled regions of interest with a virtual pen. Each time the pen was lifted, a clever algorithm closed the curve and reduced the drawing to a shaded area of the body.

This provided instant feedback to the patient and was very straightforward to anyone familiar with pen and paper. With a quick introduction by the medical practitioner, the patient was generally off and running.

Intriguingly, about 10% of the users found a second mode of interaction. Rather than circle the body region they wished to indicate, these users instead shaded it in, crayon style. The algorithm behaved predictably in response to this unplanned entry mode, providing positive feedback to the users just like a pen on paper so they continued to use it. This was a triumph of lucky design in some ways, but a big problem in others.

Data entered by shading produced vastly more closed-curve objects in a given drawing than data entered by simple circling. Our data analysis routines stepped through these closed-curve objects in a way that rendered the shaded data extremely slow to evaluate. So these 10% of our users required vastly disproportionate analysis time. We eventually wrote subroutines to combine overlapping closed curves into minimal convex hulls for purposes of both data compression and analysis speed—compensating for our own unplanned features.

The users had stepped beyond the designer's intent and found another way of successfully using the system, with unforeseen consequences. They were unforeseen because the use case variation was completely outside the design scope. It just happened to work. No amount of planned testing was likely to reveal the issue.

In this very same system a bored alpha tester also took us quite literally outside the box. The young gentleman, a university student and research intern, was asked to test our system between his other tasks. He wasn't given specific test instructions, just asked to try out different functions and let us know if anything broke. It wasn't long until he succeeded.

Asked to demonstrate for us, he proceeded to circle the entire screen several times, lift his pen, and then make one more dot somewhere else on the screen. The system crashed instantly. In fact, he had triggered a rather unlikely buffer overrun condition that we promptly addressed. This was completely outside any rational use case since there was no reason any user would ever need to circle the entire screen, let alone more than once. It took a bored lad scribbling at random to find our failure mode.

What could a bored lad achieve by punching buttons at random on your embedded control device? What unintended user interactions might be supported, even reinforced by your

user interface? You will not predict them all. Keep that in mind when contemplating the value of cross checks and critical element redundancies.

Consider next the inside-the-box elements. Today's embedded processors sport an abundance of power and memory resources and encourage increasingly complex embedded code: complex code that is less and less provable. We now need to triage our complex systems and isolate critical elements from ordinary operations code. Even so, code for

critical-system elements often interact with less reliable subsystems. Consider, for example, the now common situation of running your front-end operations and user interface on consumer-grade Windows. Even in the best of circumstances, bugs in large software systems can be insidious, with sleeper failure modes lying in wait for just the right combination of inputs or just the right intermittent hardware glitch. What about the future? If your code is subject to future modification, how resilient is it to bugs introduced later? How much testing will need to be repeated?

We simply cannot rely on high-level system testing of complex mission- and safety-critical designs. We need to fight the battle on the ground, at the foundation level of our design and coding. We begin with three strategic principles:

- Readability
- Redundancy
- Provability

The first of these principles, *readability*, will be familiar in some form or other to every software engineer who has cracked a textbook in the last 30 years. The details of the idea have evolved substantially and coding styles have gone in and out of fashion, but the concept remains and certain fundamentals persist, including clear blocking of code, transparent naming of variables, functions, and properties, extensive commenting: All of these niceties make it easier for your colleagues to read and understand your code and for you to remember what you were doing when the boss asks you to modify the code a year from now.

Example 2.2. Human Readable Is Human Recoverable

All software engineers go through a phase where they enjoy maximizing efficiency. It's amazing how much information you can pack into a binary file, coding bits and bytes in tight structures. It's also amazing how useless that information can become if corresponding "reader" software isn't available. If the binary data files become corrupted, then one can only hope the original engineer took the time to write recovery functions into the reader software. Otherwise, the hour is truly dark.

There are instances where tight data are key. Perhaps there are system storage limitations. Perhaps the data are naturally structured, such as in image or audio files. There are also instances where an embedded system is fundamentally designed to process information for compression or transmission—think mobile phones.

There are, however, many, many instances where data are being logged or recorded for testing purposes on typical modern systems with typical modern storage capacities. In such cases the additional overhead of writing an ASCII field name with the data or in writing ASCII coded

(Continues)

Example 2.2. Cont'd

text is often trivial compared to the benefits of having a human readable output quickly available to review in any handy text editor.

The Linux world has long recognized the usefulness of ASCII control files and logs. Ironically, after years of increasingly obscure file formatting, even Windows now incorporates ASCII-coded configuration and log files.

In my work in experimental test systems and embedded design, it becomes impossible to count the occasions where an ASCII log file, or an ASCII coded header to a binary data file, has saved the day. Cases include testing, debugging, crash recovery, and a few substantial projects in legacy data recovery where systems have fallen into disuse.

Not long ago I received a phone call from a researcher asking my help on understanding data that we collected with a custom test system almost 20 years ago. The system no longer exists, and even the hardware it ran on is impossible to reproduce—but this gentleman had file after file of archived data from our work. I walked him through the data file headers, written in long-form text, which fully documented the experimental setup and each data field written in the file below. Needless to say, he was rather pleased.

Documentation gets lost. Code gets lost. Processors and operating systems change. Even storage media evolves. If your data are not stored in a well-documented and broadly applied standard format and your requirements are not truly data intensive by modern standards, consider keeping it human readable wherever possible.

Readability maximizes transparency, facilitating verification and bug fixes. In the realm of applications programming, these are *good practices*. In the realm of mission- and safety- sensitive software, these are *critical practices*. When the failure of your code could result in the loss of a quarter-billion-dollar aircraft and human lives, clarity counts.

As with process, it is not my intent to espouse a particular style convention. Depending on your background, the tools you work in, and your organization's standards, your preferred style may vary quite a bit. However, general experience suggests that using as long and as descriptive a name as your own convention and software tools permit often goes a long way toward success. When code reads like a sentence, it becomes much more readable and much easier to verify. A friend who programs for a major airline-industry software provider relates the story of getting slightly embarrassing attention over her innocent choice of the polymorphic method name "isTurnedOn()" for checking the status of system objects.

However, one may rest assured that the code reviewers had no trouble in understanding and verifying the logic during the walk-through.

Keep in mind that meaningful naming conventions also require consistent spelling. In an age when modern programming languages are reintroducing untyped, undeclared, and runtime allocated variables, it pays to get a little old school when it comes to your critical system

elements. When writing code, turn on compiler flags that require explicit declarations and use code evaluation tools that help you find orphan spellings.

Programming standards have evolved through the introduction of functions and subroutines in software pre-history, to the organizational rules of *structured programming*, to the current *object-oriented* vogue. Modern embedded programmers may code in languages from assembly to Java these days, so there is not a one-size-fits-all solution. However, adapting a consistent strategy and complying with your own standards is the first step in any good practice manual.

Side Bar: Is OO the Enemy of Transparency

To the experienced embedded software engineer, there are advantages and tradeoffs to every tool. Some occasions call for hand-optimized assembly code, while others afford the luxury of the latest UML graphic design tools. Having lived through both the structured programming and object-oriented (OO) heydays, it is interesting to note that not all the purported gains of OO are blessings to the mission-critical programmer. In particular, inheritance and polymorphism can be sticky concepts since, in practical situations, progressive development of software or reuse of older code often entails modifications of root classes, potentially triggering an extensive re-verification process. If someone's safety is on the line, every test pass may need to be deep and exhaustive.

OO gurus will take exception. In fact, the same aspects of OO that allow one to define inheritance will aid in tracing dependencies and defining test plans. It is certainly possible to adhere to good OO design patterns and build systems where properties of OO work to the mission-critical programmer's advantage. In the author's experience, however, this is seldom achieved, since evolving hardware choices and other system considerations combine with delivery time pressures to spoil design perfection. Particularly in complicated event-driven object code where user input and time-cued events continually modify the system state, it becomes remarkably easy to lose track of which methods or call-backs handle which hardware checks, resets, and state updates under which circumstances. Independence of the critical safety code as espoused in the chapter can help mitigate those risks.

OO programming does have advantages. In particular, OO allows easy abstraction of real-world elements so that the code interaction model is more intuitively understandable. The "Hardware as Software" exercise in this chapter (see [section 6](#)) is intentionally sketched using an old-school, structured subroutine model. It would be much more transparent to sketch a UML diagram where an object like the "PrimaryInterlockSwitch" had a property "Closed." However, while the OO model clarifies intent, it deliberately hides functionality. Inside the "PrimaryInterlockSwitch" object, someone would eventually have to write and verify a subroutine that acted on changes in the property "Closed," and the hardware control aspects of that subroutine code would look rather similar to the subroutine presented. The UML object model must be linked to the hardware eventually.

OO may have a natural appeal to hardware engineers moving into software engineering. OO programming can be thought of analogously to component-based hardware design. In many ways, each object class is like a discrete device, with external interactions governed entirely

(Continues)

Side Bar: Cont'd

by fixed public “properties” and “methods” in the manner of the documented electrical connections on a chip. The internal manner of function is not of concern to the user; only the published behavior is important.

Relying on the hidden embedded functionality, however, dictates thorough component module testing. It also dictates isolation and tracking of critical functions—a recurrent theme of this chapter.

This brings us to another point: a key positive feature that both structured and OO programming styles share is encapsulation. Use it! As discussed in the text, encapsulation helps enable both readability and provability. However, the existence of encapsulation is not sufficient—think very carefully at the design stage about how to employ encapsulation. If safety and critical elements of the system are scattered across the system classes, the design may be a recipe for disaster. Complex interactions between classes can conceal unanticipated failure modes. Extensive, repeated testing may become necessary at each software change to verify critical system behaviors. Because of lack of transparency, verification can become a matter of probabilities rather than certainty.

How does one overcome the transparency issues that can arise in OO programming? First and foremost, by applying the principles of redundancy and provability to the safety- and mission-critical aspects of the system. Burying critical safety checks in your operational classes is both unnecessary and poor practice. Instead, encapsulate critical software aspects in independent classes that can be verified independently of other operations code.

Yes, one may very well incorporate secondary safety and status checks in operational code that are redundant with the independent safety modules; however, by clearly calling out critical hazard mitigations in the object design, the system will become readily verifiable and inherently lower risk.

The second principle is *redundancy*. This concept may be natural to the mechanical engineer designing bridges, but it is seldom second nature to the applications software engineer who relies habitually on computer hardware and operating systems to perform as advertised. It has been said that the best engineers are lazy—that is to say, we innovate intensively to save work in the long run. For many, the elegance of software somehow lies in the art of creating minimal code to achieve maximal effect. It may seem counterintuitive to save work by allowing more than one piece of code to perform the same task. It's not.

In fact we save work, in the long run, because by carefully employing redundancy we are enabling the last of the three principles: provability. In particular, we enhance provability by reducing the required testing load.

Consider what happens if you introduce a change in the operational code that mistakenly removes or inactivates a sequential safety check. Consider what happens if someone else comes along and mucks up your code with their own ideas—perhaps simply because they didn't fully understand the dependencies. No one is perfect and any practicing engineer can probably recount a few such experiences.

At another level, consider that software is inherently reliant upon the hardware that supports it. Consider how completely you want to rely on your internal state variables accurately reflecting the real-world environment if an erroneous hardware setting could permanently injure an innocent user. Hardware engineers certainly don't take their hardware's continued normal operation on faith; likewise, critical system software engineers should not take either the hardware or their own software on faith.

We address these issues by clearly calling out and enabling mission-critical and safety elements in separate, verifiable code. If the disabled in-line safety check were backed up by a redundant, independent-watchdog safety check, a potentially life threatening bug can be reduced to merely an important future software patch. If the system is able to use independent sensors to verify reality, rather than relying on internal state variables, entire classes of failure modes can be detected and avoided.

Working field software should not even trust itself. In space, military, medical, and heavy- industry environments, well-designed embedded systems always start up with a thorough self- check. Is the code intact according to checksum tests? Did it load correctly? Are there any stuck bits, registers, or sensors? Space systems, due to ionizing radiation exposure, sometimes include means of continually verifying software, processor, and memory conformity during operation—at times replicating entire embedded subsystems in two or three copies.

Hardware engineers are unashamed of redundancy because they are trained to design to failure modes. Examine the schematic in [section 6 \(Figure 2.6\)](#) and you will find that multiple redundant interlocks consume most of the design. Software engineers are sometimes burdened with an education that suggests if only they follow certain steps they should be able to achieve provable, efficient, bug-free code. That thinking is not wrong, but it is dangerously self-deceiving. Software designers need to fundamentally understand that rushed releases, borrowed code, broken updates, flakey hardware, and honest mistakes are part and parcel to the field. A critically designed highway bridge may, with some luck, carry traffic for years at its design load, but a three-times safety factor over that expected load will truly protect the health and safety of the public in the face of the unpredictable.

Assume fallibility and create failsafe watchdog checks that address not only failure of the system and user error, but also failure of the code. (See [Example 2.4.](#)) Just as in building airplanes and bridges, critical-system software must include redundancy: design a complete operational system meeting

requirements and, in addition, build in *independent* watchdog and backup elements to protect against critical failures.

Example 2.4. Brain Stimulation Backup

What happens when a little change in code mucks things up in a working system? A failsafe watchdog can really improve your day. In the late 1990s, I was involved in designing software-

(Continues)

www.newnespress.com

Example 2.4. Cont'd

controlled test systems that included electrical brain stimulation. We used the stimulation in conjunction with behavioral tasks to assist surgeons in mapping language and motor skills

in the brain before surgery. These systems were noncommercial and we frequently updated our software to incorporate new specialized tests for individual patients under very tight time pressure. Several tightly choreographed computers performed numerous operations, including presenting visual and audio cues to a patient, and recording EEG, speech, and reaction time, all while stimulating across pairs of electrodes laid directly on the brain.

During operation, the general sequence of events was supposed to work like this:

```
Sendtrialinformationtorecordingcomputer Beginrecording
```

```
Turn on electrical stimulation Present
psychological test cue Record patient
response
```

```
Endpsychological test
```

```
Turn off electricalstimulation EndRecording
```

```
Storedata
```

```
Repeatforeach trialuntildone
```

Caveats abounded. Chief among them was that electrical stimulation could under no circumstances remain active for longer than 5 seconds for safety reasons. The length of the psychological test could vary and patient response time was up to the subject, meaning that the timing and even the sequence of events were never completely predictable. The system incorporated several levels of software cross-checks and, of course, emergency hardware disconnects.

The hospital monitored human subjects work with an excellent group of independent engineers known as the “clinical engineering” group. Our job was to build it; their job was to try to break it and to make sure, in the worst case, that it didn’t hurt anyone.

As our work progressed, the physicians asked us to speed up testing by allowing multiple psychological tests to occur during a single electrical stimulation period. This necessitated a delicate restringing of the control code, and time pressures always meant too little testing. In due course, we demonstrated the updated system to clinical engineering, verifying that we could interrupt stimulation at any point using keyboard inputs and that the multitest sequencing worked properly. After an hour or so, by chance, we discovered a bug: In certain rare cases, where the last scheduled psychological test overlapped the scheduled electrical stimulation endpoint, a flag was miss-set and the state variables lost track of the fact that the stimulator was on.

It was rather stomach-wrenching to have a deeply concerned third party ask, “Why is the stimulation still on?” during this sort of clearance demonstration. However, the clinical engineer immediately checked the electrical stimulator itself and corrected himself: the stimulation was off. In fact, while the computer screen was still indicating “stimulation on” in rather bold and alarming fashion, watchdog timer code that had never before been executed in need, had already detected the 5-second timeout limit and turned the juice off. The system

corrected its erroneous display shortly thereafter, verifying reality by checking the stimulation relay status before moving on to the next trial sequence.

Perhaps this reminds you of the “new call” light mentioned in the introduction? It should. (See [Example 2.8](#).) There was no way of directly monitoring stimulation status from the control computer in this case. Instead, a function named something like `ElectricalStimOn()` checked the control relay and reported whether it was open or closed via a hardware register bit. This function did not fail; instead the operations code simply failed to clear the blinking warning from the display because the check-and-update sequence was skipped. The error is akin to the problems with setting internal-state variables to mimic reality; in this case, the presence or absence of a line of text on the display was the “flag” that a human operator saw and it was simply out of synch.

Our clinical system came out ahead of my telephone, with its perpetual “new call” situation, because of strategic redundancy: my software contained independent watchdog safety code and verified reality by forcing things to known conditions at the end of each trial.

The bug was easy to fix once detected. The failsafe watchdog timer not only saved face, it also would have fully protected the patient had one been attached. Ultimately, our third-party reviewer was very favorably impressed with the safe, demonstrable redundancy of the system, especially given the non-commercial nature of the project and the likelihood of frequent future changes.

Independently designed and tested failsafe routines that are not part of the primary control code are key to reliable, demonstrable safety. Put more simply: CYA.

This brings us to the third principle: *provability*. Certain small embedded systems are in fact simple enough to examine and prove exhaustively. These are increasingly rare, and even where they are possible, maintenance and update processes become significant costs when one must re-verify everything.

In general, operational code in modern systems is simply too complex (or too expensive) to test exhaustively. Triage in the form of RA/HA is the key to getting our hands around the problem. We can use the above principles to verify the mission- and safety-critical system aspects to achieve a failsafe product.

Some aspects of provability come about in code validation, others in functional system design. Encapsulation of mission- and safety-critical code not only aids readability, it also enables provability. For example, by designing a watchdog monitoring thread that is independent of my system operations code, I encapsulate a critical aspect of my system. That safety check can be verified in early code reviews and in independent systems testing and thereafter frozen out of most future updates. I then know with some certainty that even if the more intricate operations code is broken in future updates, I have a friendly big brother watching things. Not only is encapsulated code thread independent of the other components, making for a reliable functional design, it is also compact and easy to read and verify without tracking through the full complexities of operations.

Even where code is not multithreaded, a natural extension of this idea applies. Encapsulate the mission- or safety-critical aspect in a standalone function call that can be verified, frozen, and relied upon. For example, `MakeLaserSafe` might be a nice function name for a sequence of events that included testing for lack of energy in the exciter, powering down the laser if necessary, engaging interlocks, and warning the user if anything fails. `TestLaserSafe` might be a nice function name for a routine that checked the exciter, checked the interlocks, and set display warnings accordingly. Once these safety routines were coded, verified, and frozen, the presence of these function calls in any updated operation code would become part of the standard verification procedure.

In single-threaded code designs, even where outside physical events cannot affect hardware status, it is good practice to use these sorts of calls redundantly. In this case, redundancy enables provability. For example, one might call `MakeLaserSafe` followed by `TestLaserSafe` in the operations code as part of ending an ordinary cutting operation.

The function `UnlockDoor` might make the very same calls, redundantly, before operating the door solenoid. This would protect against later bugs in which a programmer fails to sequence the commands properly. It would also allow the `UnlockDoor` command to be verified against the RA/HA chart independently of the operations code. Conversely, it would allow operations code updates to be verified without relying on the institutional memory

of whether or not `UnlockDoor` makes the system safe.

With modern fast hardware and cheap memory, the only real excuse to avoid such redundancy is ultra tight real-time operating constraints. But generally in such circumstances the entire response-time-critical operating branch becomes a critical component subject to verification and review with each modification.

Another aspect of provability occurs during operation, but must be planned for during design. The idea can be reduced to *verify reality*.

Again, the provability and redundancy ideas complement each other. In the example above, we not only turned off the laser, we made sure it was off, hopefully by means of an independent hardware sensor. Were we to use only software state variables and believe that the laser was off simply because we turned it off, then our user might one day experience a very unpleasant failure mode. Hardware busses fail, registers fail, relays stick.

..... Safe
ty
cannot be left to an internal model of the physical system

that is not continuously updated and verified.

When software can issue a hardware command but not confirm the result, the situation is often referred to as *open-loop* control. Open-loop control leads to assumptions by its nature; and unverified assumptions are very, very frequently the root cause of high-risk failure modes in safety- and mission-critical systems.

www.newnespress.com

In a safety-critical system, your software control is extremely likely to be backed up by hardware interlocks. In fact, unless it is physically impossible to achieve, it had better be. On the one hand, don't leave home without making the hardware as safe as possible in its own right. On the other hand, do not rely on hardware safeties to compensate for your code failures. Hardware switches and interlocks alone, in some sense, work in big broad strokes, while software can generally act much more nimbly to detect, compensate for, and even predict hardware failures.

3.2. *Safety (and Mission) First*

For anyone who has wandered the grounds of an industrial lab or manufacturing plant, “safety first” seems like an overworked catch phrase. In fact, in mission- and safety-critical system design, it is an actionable design approach. This approach begins with *risk assessment* and *hazard analysis*.

Hazard analysis is a process used to assess risk. The hazard analysis identifies and quantifies risks and then defines means of controlling or eliminating them. This process occurs in the very early stages of the software development cycle, often before operational features are fully specified.

A common format for such documents is a multicolumn matrix, such as the following.

Failure Mode	Hazard Level	Design Action
(How could it go wrong?)	(How bad could it be?)	(How do we compensate for it?)

Failure modes may be caused by equipment problems, software problems, user actions, environmental situations, hardware or power glitches, or even acts of God. The hazard level typically reflects at least three levels of risk assessment characterizing the significance of the threat. This is sometimes a combination of the possible cost of the failure mode (in safety or mission risk) combined with its likelihood of occurrence. In general, however, one attempts to write down everything bad that could happen since it is often meaningless to sort out likelihood early in the design. The design action is a brief statement of how we intend to control the hazard in our design—always stated in an actionable and traceable way.

The three columns show the essence of the idea. Many practitioners develop their analysis with several more columns, breaking out information and adding additional details specific to the project or to company process. For example, hazard level might be explicitly separated into estimated probability of occurrence and risk level upon occurrence. Design action might be complemented with a

column assigning the task to a project group, and each row may include a traceable number or code to be used in tracking.

www.newnespress.com

From the perspective of software, we need to know which of the system-level RA/HA items are actionable. These items should form the basis of a software RA/HA matrix. To that matrix we should add our own additions. These include software-specific risks such as hidden hardware dependencies, potential for state verification, and software bugs that might slip through in initial coding or updates. These are exactly the topics we've discussed above.

Such hidden software risks may not always be on the system engineer's mind, and where they pose significant potential for safety or mission compromise, they should be brought to the overall design group's attention. They should also be documented and tracked within the software group.

We need to recognize that common software failures can be harder to explicitly predict and harder to identify and trace when they occur than common hardware failures (Fig. 2.3). It is incumbent upon the software engineer to recognize the limitations in our craft, to convey the risks to those who need to know, and to design and code in a way that compensates for these recognized, but often nebulous, risks.

In many systems both safety- and mission-critical aspects arise. Sometimes they are related; sometimes they are not. In medical products development, it is common to focus heavily on safety since this is what the Food and Drug Administration and other regulatory agencies will audit and control. But in any embedded software design, there will also be mission-critical aspects that are important to the success and usability of the product. A product may be perfectly safe but ineffective, rendering it a failure in its mission. Why build expensive doorstops? Safe and effective products are our goal, and we achieve this by tracking the balance of mission- and safety-critical aspects of a project.



Figure 2.3: Burnt offering: Common software failures can be harder to predict and, when they occur, much harder to identify and trace than common hardware failures. (Photo courtesy of Jeffrey M. Sieracki)

By tracking both aspects in parallel, a critical system designer—or any embedded system engineer—can make clear, traceable choices that increase probability of producing a working, verifiable system that meets design goals.

Consider again the V&V process in [Fig. 2.2\(B\)](#). After we quantify our safety- and mission- critical aspects in an RA/HA chart, we move on to designing the critical architecture components to address this list. We then design code modules to enable this critical backbone architecture. Insofar as is possible, this design will be independent of the operational architecture that we design later to meet the basic functional goals.

After implementation, we assemble our components working bottom up to test and verify. Each descending leg of the V corresponds to an ascending verification step. The critical elements detail design maps to independent testing of those modules, and the critical architecture informs the integration of those modules as well as integration with the operational elements of the system. Finally, the list of critical aspects the original RA/HA chart directly drives set of system-level validations, including both functional tests and RA/HA-specific end-to-end design reviews and code walk-through.

By starting the project with the critical elements rather than general operational aspects of the project, we design to the failure modes. We can then move forward on the feature-driven aspects of the project with some confidence that we have covered ourselves on the critical side. In the author's experience, this safety-first approach often leads to clearer and simpler operational code for two reasons: (1) the operational code can be written directly to the feature list without thinking and rethinking what might go wrong at every step; and (2) the critical design architecture often becomes the backbone of the basic operations architecture with very little additional overhead.

Use cases can also be extended to failure modes. Modern software designers often build up a collection of storylines that helps all stakeholders visualize how the system will interact with users and the environment. These are called use cases. They are generally used to help define user interfaces and operations sequences to enable the system to better serve its application. In practice, their use is typically emphasized in early design stages to help flesh out the look and feel with respect to the basic mission; their importance generally diminishes as development progresses and the actual system elements are available for examination and testing. Application of use cases is often limited to understanding and defining core operations elements.

However, they can also serve as a powerful tool for evaluating failure modes and in planning for risk mitigation in safety- and mission-critical systems.

What happens if an alarm sounds in my plant control center or on my aircraft cockpit panel? Is there a defined user checklist? Are there mechanisms by which a user can acknowledge the situation and eliminate the distraction? What if the user simply silences the alarm and moves on—is there contingency code for ensuring that critical alarms cannot be ignored?

www.newnespress.com

Suppose that a primary control fails. Consider the steps by which the user (or the embedded software) will enable a backup system. Is it natural and achievable under expected operating conditions? Are additional steps required to ensure safety during transition?

Consider the myriad insidious software errors discussed earlier. Can the code help a user recognize possible internal failure? At a minimum, we should log inconsistencies in internal states variables and sensors so that they can be reviewed later. Failure of a hardware system to respond as expected could be a hardware error or a software bug. Where detection is possible, it should be addressed automatically to the fullest extent possible.

The failure-mode use-case analysis should assess the risks associated with the problem, whether and how the user will be notified, and how to log the error and notify technicians that something must be investigated. The “check engine” light in your car is a classic example of dealing with low-risk failure modes—by simple means it notifies the user that a non-critical problem exists that cannot quickly be acted upon. The actual problem is logged and a technician can later short a few jumpers to request that the embedded software report the problem in detail.

In some cases, the failure will present a clear and present hazard to safety or to the critical mission of the system. The failure-mode use case should carefully address how the user is notified, and whether and how the system is shut down, the mission aborted, or the situation is otherwise made safe as appropriate.

Example 2.5. Expecting the Unexpected

We give a great deal of emphasis to risk assessment and hazard analysis, as, indeed, we should. These critical paths become part of our design and traceable elements in our test plan.

Similarly, requirements documents and specifications lead to our core operational design and operational test plan.

In both commercial and research systems requirements, there are always requirements that will not be documented in advance. These include tacit assumptions or environmental factors that do not become apparent until we get to field releases and beta testing.

In developing a computer-based medical research system designed to be left alone with a patient, we did not fully consider the habits and expectations of the clinical staff. Commercial medical devices are nearly always either permanently installed, equipped with battery backup, or both. Nurses thought nothing of walking in, unplugging the computer, and wheeling it out of the way. In the days before uninterruptible power supplies became cheap and ubiquitous, this was a serious problem. Hours of patient testing could be lost if the system was not shut down properly. Thus “plug-pull tolerance” made its way to the top of the evolving requirements list after a few weeks in the field. By ensuring that the software flushed file buffers early and often and that we tracked recovery information in case of sudden interruption, we prevented loss of data. This meant less wasted patient time and fewer surly exchanges between the

To the surprise of the surgeon who directed the lab, the change was fairly quick and easy. The change was accomplished with minor, traceable edits to the working code and one small additional module. There were three changes: (1) The file system was flushed after each minimal unit of meaningful data was collected. This required a single line of code and did not impose any unacceptable delays in operation since it was coordinated with a natural momentary break in the test sequence. (2) An indicator flag file was written before patient testing began and cleared after testing ended. (3) On start-up, the software checked for the flag file and, if present, executed a separate subroutine that checked the last data file for any partially written information and cleaned it up. The software then passed parameters to the existing code to restore the patient interface to the point at which the plug was pulled.

The code was already well encapsulated. The only modification to the critical core control code was the file flush—a single line of code, easily verified. The other modifications were outside the core control code and added to the start-up and wrap-up sequences—again with one or two lines of code, in clear sequence, easily verified. Existing well-tested software modules already worked reliably and major modification and retesting was unnecessary. The new clean-up subroutine was created and verified separately.

Inevitably, your requirements and your code will progress, so encapsulate and generalize safety- and mission-critical aspects. Thinking and designing modularly is efficient at every step, from documentation to code delivery to testing.

In addition to failure-mode use cases, also consider the postmortem in the event of failure. Consider how the technicians can learn what went wrong so that they can repair the system, or convey issues to the designer to fix in the next update.

With respect to good design practices, we can learn a lot from the black boxes used in aircraft and other public transportation. At the hardware level, they follow all of the mission-critical design guidance steps that we are advocating for software. The boxes comprise an independent, encapsulated, and separately tested subsystem. The encapsulated design is limited to a specific, narrow critical mission, and is not subject to revision in conjunction with the rest of the operational system.

Black boxes have become ubiquitous in transportation and are, to some extent, becoming ubiquitous in applications software—in the latter case, they take the form of system event logs. In embedded systems, they have often been limited to simple error-state flags, such as the “check engine” model. With increasingly cheap computing and memory capacity, the safety- and mission-critical designer has little excuse not to include a means to keep detailed track of field problems.

3.3. Verification and Redundancy in the Implementation Process

Relying on encapsulated functionality, as discussed previously, dictates thorough component module testing. These critical paths become part of the design and should be traceable

elements in the test plan, just as requirements and specifications documents lead to the core operational design and operational test plan.

To the extent possible, meet specific needs with specific traceable code. In some cases, this may be a watchdog timer running on a separate thread; in others, it may mean encapsulating hardware and sensor interactions; and in still others, it may be as simple as laying out code so that a master loop clearly and unavoidably checks and confirms hardware status on each pass-through.

Ideally, each software mitigation step listed in the RA/HA should be traceable to a specific, encapsulated code segment. This will reduce risk of unforeseen interactions, simplify verification, and increase the likelihood of delivering a robust, operational system.

In practice, not every aspect of every mitigation step can be encapsulated. For example, start-up code needs to make calls to initiate certain modules. Operational sequence code may also need to make specific tests to take advantage of the encapsulated aspects—such as making the laser safe before opening the door. As discussed earlier, however, a certain degree of redundancy can help simplify verification and ensure compliance with required procedure.

Non-encapsulated steps need to be conveyed clearly to code designers and verified as a matter of course in code reviews. Permitting and encouraging strategic code redundancy where safety- and mission-hardware interaction occurs not only aids the reviewer's work, but also provides a safety net. It is certainly possible to achieve Six-Sigma level verification without doing so, but the potential for hidden bugs in complex modern software is so high that the belt-and-suspenders approach of targeted redundancy will almost certainly be a safer and more cost-effective plan.

Targeted redundancy is not synonymous with bloated code. Repeated elements are limited to calls that verify critical hardware status, and the redundancy occurs in that these calls may occur in more than one subroutine that may be part of a sequential code sequence.

The hardware functions should be encapsulated and the operational code clean and readable. Old-school software efficiency hounds (the author among them) can take solace in the reality that modern chip designers often must include a high percentage of entirely redundant circuit regions on their silicon dies to compensate for frequent failures in the manufacturing process. A high mission success rate takes precedence, and carefully applied redundancy is ultimately efficient and cost effective.

Version control, test and control plans, and linked verification

of the elements in the RA/HA matrix are the means by which safety- and mission-critical elements are traced and ensured in the final product. The process documentation often feels like a burdensome evil when an engineer would rather get down to coding, but it is ultimately a powerful tool for quality assurance. Taken with a positive view, the process can be leveraged by the critical-system engineer to make implementation a very targeted and efficient process.

www.newnespress.com

Other elements of redundancy are becoming standard fare in modern development.

A generation ago, programmers tended to be quite curmudgeonly about their peers looking over their shoulders. Today code reviews are ingrained in the development process. By subjecting each function call and line of code to multiple pairs of eyes, the chances of catching and identifying bugs and obscure potential failure modes go up enormously. Code reviews should consider style, insofar as keeping code readable and conforming as necessary to maintain consistency across a work group. However, the emphasis of the review should be on examining whether a particular code segment meets its design purpose and avoids hidden flaws. The best code reviews include multiple team members stepping through the code visually, not only looking for bugs but also challenging each other intellectually with various use cases and code entry conditions to determine whether something was missed.

Adding in safety—or mission—critical crosschecks can become natural. If reviewing encapsulated critical code, make sure that it meets its design parameters, that it is tight and straightforward, and is strictly independent from other systems operation insofar as possible. If reviewing general operational code, have the RA/HA checklist handy, ask risk-associated questions as you go, and make an explicit pass through the checklist once to evaluate that each condition is either addressed or not applicable.

Some shops have introduced more extreme levels of implementation code production redundancy with good effect. These include ongoing multiple programmer integration and even paired “extreme” coding teams that work side by side on every line. The success of these ideas lends more and more credence to the cost effectiveness of carefully applied redundancy when it comes to saving time in achieving a reliable result.

4. The User Interface

On June 22, 2009, a DC metro train at full speed rear-ended a second train that had stopped shy of the station. Six people were killed, and over 50 injured. The failure was not supposed to be possible. Not only are the trains generally operated under automatic control, they also have a separate system that automatically stops a train if it enters a track zone already occupied by another—even when under manual operation. In theory, the system was designed so that if any critical sensor or communications failed, it would “fail safe” by bringing the affected trains to a halt. Clearly something failed out of keeping with anyone’s

predictions and was either not detected or not acted on by the critical safety systems. The ultimate backup control system—the human operator—also failed to notice or act in time to prevent disaster.

Today even jet aircraft can demonstrably take off, fly, and land themselves without human intervention. For an airline pilot, or a train operator, a boring day at the office is an excellent state of affairs. One reason that systems like this still have human operators is to provide a

www.newnespress.com

sanity check on the state of the environment. Our best efforts as designers can never foresee all the possibilities. In spite of all possible engineering finesse, a component will eventually fail or an unplanned situation will eventually arise.

When something does go wrong, letting your user know about it is critical. But just as important is keeping your users informed of the state of things in the ordinary course of operation in a way that is meaningful and actionable. An uninformed user may act unwittingly to cause a problem; conversely a user overwhelmed with information cannot discern and act promptly on important data. Alarms and alerts must also strike a balance. Users can get bored and inattentive in a vacuum, but frequent, low-priority alarms lead to complacency and bad habits.

Numerous transportation and heavy industry safety failures have been attributed to users either inappropriately disabling or sometimes literally fighting with safety systems. This has happened with stick-shaker and other incrementally introduced safety systems in aircraft. In some cases, automated systems can inadvertently hide information. Again, in aircraft, several cases have been documented in which an autopilot system has progressively compensated for a problem, such as flight surface icing, until it either fails suddenly or is manually turned off by a pilot who is instantly thrust into an unexpected critical situation.

Failures have also been attributed to simply having too many features and not enough information. An example of the second sort occurred in the 1990s with an implanted medical device designed to deliver dosed concentrated pain medicine directly to the spinal column. A laptop style controller was used to program the pump system via an RF link. The user interface was seemingly straightforward, with doctors able to simply tab down and adjust a list of operating parameters, such as on–off time, dose rate, and so forth. There are wide variations in delivery rates for different drugs in different circumstances and the device had multiple anticipated uses. For flexibility, the software allowed the users to change the units in which dosing was measured. However, the units were not adjusted as a matter of course and were not on the main operating screen. This led to a situation in which it was possible for an unknowing operator to program a dosage that was off by factors of 10 or 100 from his or her intention—a possibly life-threatening error. In this case “feature creep” had dangerously outpaced hazard analysis.

It’s easy to provide information in software, but good user interfaces take hard work. Logical organization of the information and triaging of critical data are extremely important aspects of safety- and mission-critical software design.

Designers of high-performance aircraft were among the first to stumble down this road. Look at the “steam gauge” cockpits in the F-4 Phantom, 1960s jetliners, and most subsequent high-performance aircraft right up until the 1990s. More and more systems were added, each with its own control quadrant and display system. Training on new aircraft became more a process of learning to find and process information than it was of

www.newnespress.com

learning to handle the new flight characteristics. First multifunction panels, then critical- information—concentrating heads-up displays, and now modern glass cockpits have all been exercises in clearly using software to *reduce* information and convey knowledge rather than raw data.

Radar and navigation systems designers journeyed down a similar path, adding more and more elements to their display systems to an overload point where buttons literally labeled “de-clutter” were then added to the design. Next-generation designs took a general-purpose engineering approach, giving the user menus by which to select what elements to show or hide on the display. The newest systems are finally taking effective advantage of software by triaging what is displayed by its importance and by situational context (e.g., zoom in when approaching destination). This gives the user “soft” control over the situation, so that they can customize their experience without inadvertently shutting out critical information.

It is impossible to generalize every aspect of critical-system, user interface design. A jet aircraft, a satellite phone, and a nuclear power plant all have very different control and display requirements. However, there are a few critical common factors to all user interface design.

The first of these is efficient information conveyance. We have already touched on several aspects of efficient information conveyance in the examples above. It is important to concisely and clearly convey situational knowledge to the user rather than simply raw data. In some cases, this means designing effective display tools—a graphic speedometer or thermometer conveys information much more rapidly and concisely than a screen full of digital numbers.

In many cases it means combining information to convey situational awareness. A graphical position on a map with an error ring is far easier to process than longitude and latitude or range data from five separate systems. A single indicator that says “laser safe” is sometimes far more functional than a screen full of subsystem status information.

It is critical to work with your target audience to determine what information they need to know in what circumstances, and in what form they would find it most useful. Starting with what is already familiar is a safe bet, but don’t fix on preconceived ideas. As designers and engineers, we are full of clever ideas and possibilities. Some of these may, indeed, be the way of the future; all of them need to face the cold light of day in actual user field-testing.

Responsiveness is another significant factor in both user

experience and safety- and mission- critical effectiveness. Human reaction time can range from a few hundred milliseconds to over a second, depending on the task, age, and physical condition of the subject. However, we are anticipatory creatures who expect our interactions to be regular and predictable. Poor user interface experiences can degrade safety and increase mission risks.

www.newnespress.com

Consider my cell phone—being a “smart” phone, it has a lot of capabilities and processing capacity. But this capacity is noticeably finite. The limitations become all too evident during boot-up. While the O/S merrily winds up, the phone begins searching for a signal, loading system components, and otherwise verifying its existence. The designers gave some modicum of thought to priority levels, reasoning that when voice communication is your mission, painting pretty pictures can wait. Hence the display is perceptively very low in the queue servicing. Trying to operate most functions during the first minute or two of spool-up is not a good experience—inputs are buffered and multisecond display lag means it takes great care and planning not to get lost in menu navigation. This is simply a hardware limit, but the designers were in one aspect thoughtful about this limitation.

While navigation is slow, numbers entered on the keypad are nearly immediately reflected on the display. This means that even while the smart-phone features are bogged down and unusable, I can still make a basic phone call as soon as I have a signal. In this case, the responsiveness has been reasonably tailored to the critical mission: making phone calls.

Consider what information and control inputs are critical to the safety or mission of the system. Make choices that make it possible to meet these needs. This may seem obvious from the perspective of real-time system design: You must service priority elements in a time- deterministic fashion. But it is not always a leap that designers make with respect to the user interface in other circumstances.

Even when the mission is not hard real time per se, the system may include display elements that are time critical. These include warnings and alerts, as well as other safety or mission-critical situational awareness. In addition, general display lags can contribute to safety or mission failures simply through increased user error and frustration.

Flexibility can also be dangerous. As with the medical drug pump example, hidden information and diverse user options can lead to unsafe operation. In adding features, it is important to discern not “what can you do” but “what should you do.” What features and display options will enhance the performance without compromising the core mission, compromising safety, or potentially misleading users into inappropriate complacency?

Each option demands its own risk assessment. If the option carries substantial risk and is unnecessary to operations, rethink it. If the option carries substantial risk and is necessary to operations, control access and clearly flag the dangerous

operation mode.

For example, the drug pump “units” feature could have been placed in a restricted access menu, requiring a password to change. The units could have been flagged clearly on the usual operating screen, and called to the attention of any operator when not in a standard setting. Even better, the software could have examined other settings, such as the pump type and medication type, to provide a sanity check and flag unusual circumstances.

www.newnespress.com

Consider a train system with automatic proximity braking designed to avoid collision. Clearly there will be circumstances under which the train may need to be operated closer to another train than the system would normally allow—perhaps in joining cars or during testing and maintenance cycles. Access to such an unsafe mode should be controlled, possibly with a physical key system, and warnings of the unsafe mode should be clear and persistent.

This leads us to alarm clarity. Problem notifications should be clear, accurate, and above all actionable. (See [Example 2.6](#).) The user must understand immediately which alarms are advisory and which alarms are critical. Alarm clutter is a serious cause of safety issues. If an alarm becomes routine, it is not an alarm. Users become complacent and may fail to heed the same message when it is critical, or even fail to heed other messages because they are simply conditioned to clear the alarm and ignore salient information.

As mentioned previously, triage is important. Design and RA/HA review should consider how to rank and rate alerts and alarms. Those that are merely warnings can often be flagged clearly but less obtrusively so that users are not conditioned to ignore more significant alerts.

For example, rather than announcing when a non-critical sensor reading is out of range, it might be appropriate to merely highlight the problematic data field in another color. Some successful systems avoid alarm clutter by simply announcing that a non-critical warning exists and leaving it to the user to actively investigate the warning at their convenience by going to another screen.

Determining whether an alert is actionable can play a key role in triaging the event. If the user can do nothing about it, the detection is merely a data point, not an alert. On the other extreme, if the event is so mission- or safety-critical so as to demand instant action, the designer needs to carefully consider whether the system should automatically take action rather than merely begging the user for attention. The most important alerts are those that are safety- or mission-critical and require immediate human action.

Example 2.6. Error Messages Matter

The ubiquitous “check engine” light on the modern automobile has frustrated many an engineer. Why give such incomplete information? The simple answer is to not provide too much information to a user to whom it makes little difference. The errors indicated by your check engine light range from computer errors, to sensor failures, to timing problems and emissions control issues. None of these are correctable by the driver in the short term, nor are they critical enough to demand an immediate change in behavior.

Other “idiot” lights are more specific: “Check oil” demands almost immediate action. “Brakes,” “hot,” “gen,” and “low fuel” demand fairly prompt attention. Give users what they can act on. In aviation, the critical notice is often reduced to “in op,” simply red flagging an instrument

(Continues)

www.newnespress.com

Example 2.6. Cont'd

that can no longer be relied upon. The best notices are useful, actionable, and lack hidden assumptions that might cause an inexperienced person to aggravate the situation. Good user interface design anticipates the likely audience.

Give the user information that can be acted upon. Otherwise, give him or her an indication that service is required at their convenience, such as “check engine.” Leave it to the technician

(or the engineer with a good mechanics manual) to collect the detailed information and correct the subtler problems under controlled conditions.

Software in embedded systems is increasingly reliable. This was not so in the early days

of building on-off systems on available hardware such as the IBM PC. In addition to detectable failure modes and code exceptions, these systems would also fail catastrophically because of charming idiosyncratic compiler bugs, operating system errors, or actual hardware burps.

Thus it was important to include a catchall exception handler in the code so that the technician could gather information about what happened and try to avoid repeats.

In developing an experimental PC-based medical test system for controlling spinal cord stimulation, I came on just such a situation. The exception handler grew and grew to accommodate and recover gracefully from known issues; nonetheless, when a new error appeared, it needed to be tracked and fixed. From an operational use case perspective, this usually meant shutting down testing and starting over in a controlled fashion. In order to avoid startling or confusing the patient, we did this gracefully with the equivalent of a “check engine” light.

We have all made a few regrettable word choices, and here I must admit to one of mine. Imagine if you will, that you are a patient with an electrical device implanted in your body and you are left alone in your room to work with the computer. You are physically wired to the machine with an RF antenna lead taped to your lower back. You are experiencing strange sensations from the implanted device and probably a bit nervous about the whole situation. Things seemingly are going well when, suddenly, the screen goes blank and then announces, “Fatal error. Please call for assistance now.”

In the era of the classic Microsoft Windows “blue screen of death,” we had done them one better. This message occurred exactly once in the field and, fortunately, the patient was computer savvy and very understanding. However, upon consulting with the medical staff, we rapidly changed that particular message to “*system error*” and released a new version.

5. Rolling Your Own

Whether there are tight real-time requirements, good embedded control systems share certain common properties and design aspects. You will benefit from understanding common patterns, and even to rely on an existing legacy systems or commercial RTOS. (See [the following side bar.](#))

Side Bar: RTOS or Not RTOS

An RTOS, or *real-time operating system*, provides standard elements of embedded system operation that facilitate building real-time systems. The RTOS will typically include multitasking with minimal thread switching latency and give developers tools to guarantee deterministic timelines. These advantages can become stars in real-time operating situations.

But one person's "real-time" requirement is another's Sunday walk in the park. Human interaction typically demands multimillisecond response times, with, say, fly-by-wire flight controls requiring a tad tighter time cap than cell-phone display updates. In fact, one often speaks of *hard real time* versus *soft real time* in practical applications. Typically, hard real-time requirements exist when late completion of a task may lead to critical failure or useless results, while soft real-time requirements exist where the system can tolerate latency while degrading gently.

As illustrated by the mobile phone in my pocket, it is often acceptable to delay informational updates on the LCD in favor of boot-up and call processing threads. This does not make me a fan of the designer from a user-experience point of view, but I admit that I would rather have a slow-moving cursor than audio drop-outs and missed calls. In contrast, a fly-by-wire flight control may also degrade gently, at least over the span of a few milliseconds. But pilots operate to some extent by muscle memory and increasingly long or unpredictable dynamic response times will rapidly put the pilot behind the aircraft and degrade flight operations to a high-risk condition. Likewise, in a more down-to-earth example, I would not want my car's anti-lock brake controller waiting for me to adjust the radio.

Car-engine control systems, medical devices such as pacemakers, and systems that include software signal generation are other examples of systems that typically have tight, hard real-time requirements. Not all operations will have the same tight requirements. A good RTOS allows us to define priorities among threads and embed hard deterministic time limits and orders of execution.

So why would you not use an RTOS?

First of all, not all mission- and safety-critical systems have critical real-time aspects. Often, those systems that do have limited, critical time requirements can offload the requirements to separate components—either independent software or hardware.

For example, in working on a spinal-cord stimulation control system, we isolated the timing-critical elements in a separate subsystem. This embedded control device ran simple, tight code that served only the hard real-time aspects of the system, such as inter-pulse timing, energy adjustment, and emergency shutoff. The bulk of the system was soft real-time and resided on the controlling PC running a standard O/S. The O/S provided convenient tools for extensive user interface development, data storage and processing, and provided platform portability. The control system provided safety checks on operational parameters and included mission-critical functions, such as reliable interaction with the patient; however, none of these had hard real-time requirements that were not redundantly backed up by the separate controller subsystem.

This brings us to convenience. If your requirements dictate heavy user-interface development, a small footprint RTOS may not be the best starting point. If your requirements dictate use of

(Continues)

www.newnespress.com

Side Bar: Cont'd

broadly available “standard” platforms for deployment, then an RTOS simply may not be available or practical to install. Licensing and royalty aspects may restrict your company’s willingness to invest. RTOSs, like any other third-party components, are subject to changes and discontinuation, thereby adding elements of project risk. Finally, a full-blown RTOS may be overkill for many applications.

Also be aware that multithreading and interrupt-driven operating systems by their nature can introduce a host of other verification issues. For example, so-called *race conditions* may arise in which one process is unexpectedly critically dependent on the sequence or timing of other events. Such issues can occur where hardware responses or user input happen with variable time course on competing threads—they comprise one more reason to emphasize keeping safety-critical system code encapsulated and deterministic.

Some engineers designing smaller embedded systems or working on nonstandard-platforms decide to “roll their own” with respect to the core control code. This is sometimes due to lack of familiarity with the RTOS options; but often it is a carefully considered design choice balancing the system’s real-time needs, interaction complexity, investment costs, and learning curves.

Classic time-sharing designs switch between task threads on a regular round-robin basis. When the entire system is yours, time-sharing designs are generally switched on a periodic clock interrupt. Each thread is programmed as if it has sole use of the machine and interaction is based on semaphores and process blocking.

More modern event-driven designs switch threads based on priority, with the occurrence of a higher-priority event preempting a thread in progress. The approach is natural, since many embedded systems respond to real-world dynamics; however, it requires very careful consideration of interrupt interactions, since a process requiring service can make only limited assumptions as to the state of the system or the sequence in which it was triggered.

On very limited processors, it is quite reasonable to simulate aspects of these O/S design philosophies with simple tight loops and switch statements. Consider the flow chart in [Fig. 2.4](#). After a start-up process, the main loop executes continuously until the user shuts it down. Each time through the loop, the process checks the status of a list of system-state variables and acts if they are true. This simple loop is the equivalent of a round-robin list of tasks with equal priority. Setting a flag variable blocks or unblocks a particular task and allows operation. Each of the side-chain processes act as threads that can perform tasks or can set and clear request flags. These system threads interact primarily by setting state and parameter variables that are examined and acted on in the next pass-through.

Provided that no piece of code traps execution, this simple design can actually deliver deterministic real-time behavior. The danger, of course, is that without preemptive interrupts,

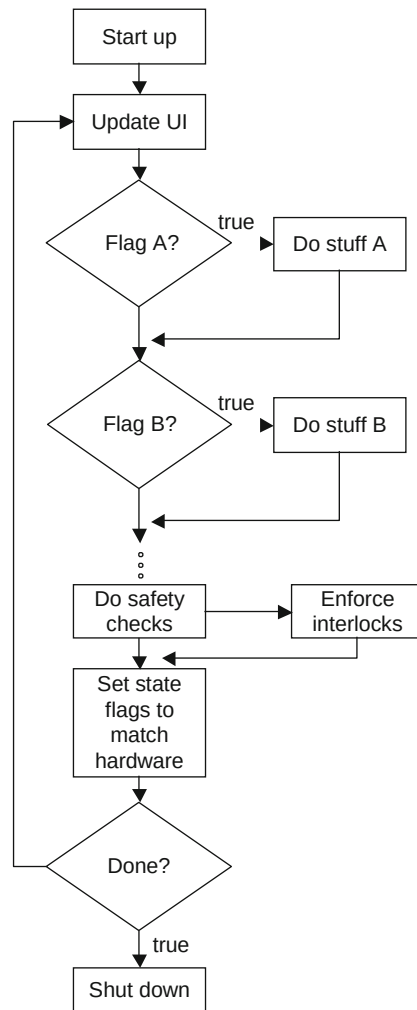


Figure 2.4: A typical tight-loop, switched operating design.

a bug or unexpected condition in the sequential code may lead to lockup. The design also lacks flexibility in task ordering.

We can make this design pattern more flexible with dynamic scheduling. In [Fig. 2.5](#), we replace the fixed switch code order with a tasking process that acts on a sequential list of scheduled events. On each pass, a new event is pulled from the list and the queue trimmed (or a pointer advanced). Switching code processes each active event; this code executes one or more subroutines to handle the event and perform requisite functions. Interaction between task threads can still include semaphore flags; however, these task flags can also modify the queue by adding, subtracting, or resequencing upcoming event requests.

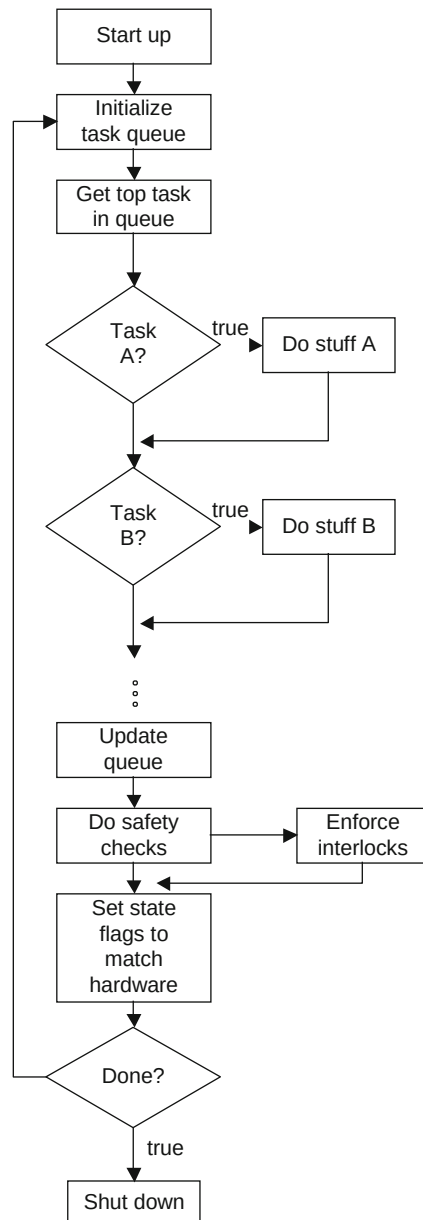


Figure 2.5: A typical tight-loop, queued operating design.

Note that including parameters for the queued tasks in the same data structure is only a small addition, so that the system can vary details of scheduled events. One of these parameters may even be an execution start time, so that events are delayed and left on the queue until a system timer value exceeds the passed parameter. Depending upon implementation, the “update queue” process may be as simple as advancing a pointer or clearing the top item;

however, it might also include a quick sorting operation that tests requested execution times of newly added items and ensures proper priority sequencing.

In practice, the actual switching code will depend on the programming language and the processor: It may be a switch statement, it may be a list of conditional tests, or it may be a jump to an executable memory location. This detail is not important to the design pattern.

Scheduled events will typically include most operational tasks. Interactions can be quite complex: One event may schedule another event, change operational parameters, or remove one or more events from the queue. Note that one can easily schedule and queue some of the same tasks that are in the round-robin test list—taking advantage of the independent encapsulated code to handle both cases.

This brings us directly to the applicability of this design pattern in mission- and safety-critical systems. As shown in the figures, the fixed round-robin event section is a natural place to embed safety checks and interlocks. They are always executed, on each loop, no matter what task is processed from the queue. However, nothing prevents a designer from calling one of these safety checks, interlocks, or other operations within the operational code during a queued task. This permits re-application of heavily tested safety-critical routines that have been established early in the design process. It provides redundancy at the level of code reuse as well as critical crosscheck redundancy at the level of system operation. Memory overhead is minimal since critical code is encapsulated in subroutines. Reviewers can check each task for safety steps and separately confirm that the main loop always contains backups.

If one considers the scheduled events to be ordinary operational processes, it becomes natural to look at fixed round-robin tasks, such as safety checks and hardware state verification, as the equivalent of “interrupts” in a more advanced operating system. These conditions are tested on every loop with finite time latency. If true, they are acted upon. As with true hardware interrupts, good design dictates that these fixed events are quickly executed to minimize delays before returning to ordinary system interaction.

In some cases, rather than executing a long subroutine, a fixed event might instead test a hardware condition and then queue an appropriate action event to process the information through ordinary scheduling. Judgment must be made as to where to encapsulate the safety-critical actions, making sure to include those subroutines in the critical path testing, verification, and freezing process during development.

This section is intended to provide guidance on how one might incorporate fail-safe design patterns, even in very basic, homegrown embedded software. If your embedded system incorporates an O/S that already provides much of the event-handling architecture, the key failsafe ideas of encapsulated independence and redundancy should be retained and adapted to that structure. The choice of whether to rely on such solutions or to turn to a full-blown RTOS that handles deterministic queuing and multithreading is up to the engineer.

www.newnespress.com

6. Hardware as Software: A Thought Exercise in Crossover Thinking

Both hardware and software engineers can benefit from crossover thinking when it comes to failsafe software design. Let us consider the microwave oven as a tangible example.

We will start by considering the hazards and failure modes, look at how they are typically addressed in hardware, and then, as an exercise, consider how we would mimic the hardware design in solid traceable software. One would *never fully eliminate hardware interlocks*

in a production appliance, simply because hardware can always fail in a manner that is undetectable in software.

However, this thought exercise illustrates good practices in design that can be applied where software is required to play a major role in a mission-critical system.

First, let us carry out an abbreviated risk analysis of our basic microwave oven.

Failure Mode	Hazard Level	Design Action
Line power surge	Low	Inline fuse
Internal short	Moderate	Inline fuse
Oven cavity overheat	Moderate	Thermal protection
Magnetron overheat	Moderate	Thermal protection
Energized with door ajar	High	Door interlock switch
Interlock switch failure	High	Interlock monitor switch
Interlock double failure	High	Secondary interlock system Disable control logic Warn user about door ajar

In addition, our usability requirements will include deciding under what conditions to allow the operations control logic to operate, and when to make an error condition apparent to the user so that it may be resolved.

[Figure 2.6](#) illustrates a realistic hardware circuit that addresses the failure modes above. It is somewhat simplified for this discussion, but you will find that ovens in many kitchens comply with this basic model.

In brief, a fuse and two thermal protection devices cover the “low” and “moderate” risk- failure hazards by providing direct interruption of the primary line power to the device.

Energizing the magnetron with the door open is an extremely high-risk failure and is addressed with redundant interlock systems.

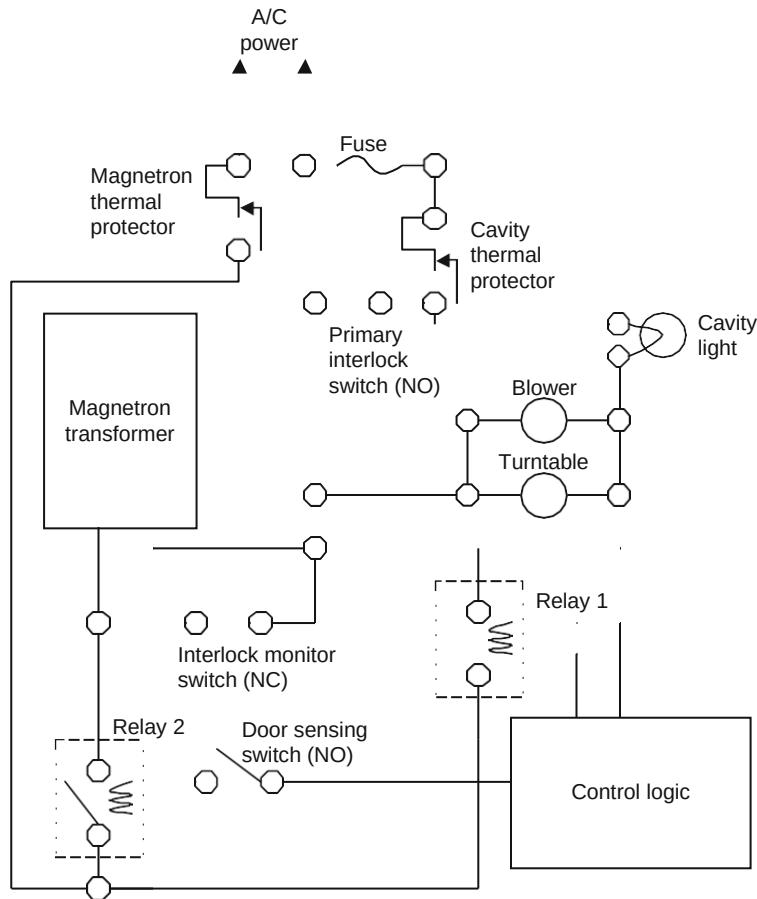


Figure 2.6: Typical schematic for hardware interlocks in a microwave oven.

The primary interlock switch is a mechanical door interlock that is open while the door is ajar. It prevents line power from reaching the magnetron transformer with the door open. Because it is possible that this switch could fail closed (perhaps due to age-related failure or simply dried food wedged in the wrong place), there is a second interlock monitor switch that shorts the transformer while the door is ajar. These two opposite acting physical switches provide primary protection against software errors, wiring shorts, physical abuse, and other significant failure modes.

It is possible, however, unlikely, that both of these switches might fail—one fused shut and the other stuck open. Low probability aside, the results would be very high hazard so the engineer has also incorporated a secondary interlock. This is accomplished with another,

physically separate door-sensing switch. The sensing switch informs the control logic unit of the door state. In addition, relay 2, used by the control logic to energize the magnetron under proper conditions, is physically wired through this same switch, providing a final backup to software failure. Note the unabashed redundancy.

In ordinary operation of the oven, the control logic software, sensing the door ajar, disables keypad input so that (1) the start button cannot be activated and (2) the user has feedback that the oven will not function. Additional user feedback is provided by the cavity light, which remains lit until the door is closed. This provides an indication to the user as to why the keypad will not operate; more elaborate software might augment the door light by displaying “door ajar.” The error indication is clear and actionable.

There is significant redundancy built into this system. In fact, we could eliminate every mechanical switch but the door sensor and the basic features of the oven would operate correctly. Even though the boss might like to save a little money on switches, it is unlikely that a hardware engineer would be considered wasteful. The liability for a field failure is huge. Likewise the critical system software engineer should not look ill upon a few extra lines of code that add clarity and resilience to the design.

Note the labeling of the components. We consider the direct hardware interlock as the *primary* failsafe protection, and the ordinary operational sensor as the *secondary* system. Many software engineers not versed in mission-critical programming would address this system design in the opposite direction. That is, they would start at ordinary control operations, and then consider detection and notification of error states, and finally consider how to address failures outside their control. This forward approach is dangerous because the programmer cannot help but tacitly assume that his or her code will basically work correctly and that it can and will be fully tested and debugged. Yet forward thinking is entirely natural to the engineer because the control logic and the operational design features are the satisfying part of seeing a project come to fruition—the fun part of software.

Instead we make “safety first” our design approach. We give some real thought to “when all else fails,” even before spending much time on the “all else.” Then, with basic safety fallbacks in mind and set aside from the operational aspects of the project, we move ahead to the more challenging feature sets.

Let's begin. We will build a basic code structure that mimics our microwave schematic while using principles of testability and encapsulation to isolate and control risks. For purposes of the exercise, we assume that we can replace each circuit-switching device with a software- controlled relay and a sensor. The schematic in [Fig. 2.7](#) shows the idea. Each sensor can

www.newnespress.com

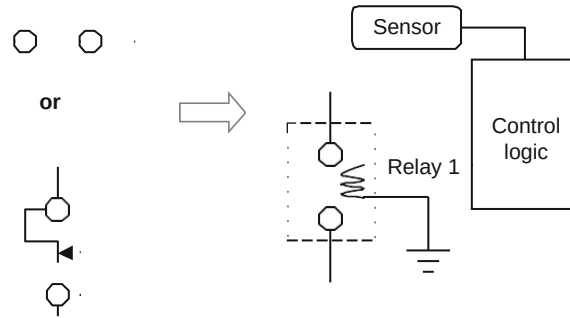


Figure 2.7: Imagine replacing each physical switch with a software-controlled sensor/relay combination.

be read and each relay set directly by control logic via a simple applications programming interface (API).

Such APIs are typical and there is no sense dwelling on the bit-twiddling aspects of this example. More to the point, there should be no tolerance for propagating arcane code lines in a mission-critical project. So we begin by encapsulating our hardware operations in meaningful functions. In pseudo-code, our basic operations for the primary interlock system are:

PrimaryInterlockSensorDoorClosed()

```

if(primaryinterlockdoordetectsensorregister)           ¼ true then
    return true
else
    return false
endif

```

to read the primary interlock door sensor,

PrimaryInterlockRelayClosed()

```

if(testprimaryinterlockswitchstate)                     ¼ (closed) then
    return true
else
    return false
endif

```

to check the primary interlock switch relay status,

```

SetPrimaryInterlockSwitch(setstate) if setstate true
then                               ¼
    (setprimaryinterlockswitchclosed) wait(delay)

```

```

    ifPrimaryInterlockRelayClosed()      1/4 truethen
        returntrue
    else
        returnfalse endif
else
    (setprimaryinterlockswitchopen) wait(delay)
    ifPrimaryInterlockRelayClosed()      1/4 falsethen
        returntrue
    else
        returnfalse endif
endif

```

and to set the primary interlock switch. Similarly we can define other basic operations. These are simply listed in the following:

SetInterlockMonitorSwitch(setstate)

Tocontroltheinterlockmonitorswitch

DoorSensorDoorClosed()

Toreadthesecondarydoorsensor

SecondaryInterlockRelayClosed()

Tocheckthedoorsensorrelayswitchstatus

SetSecondaryInterlockSwitch()

Tocontrolthedoorsensorrelayswitch

SetMagnetronEnergizeSwitch(setstate) Tocontrolthe

Magnetronrelay2

The exact naming is not critical, but we emphasize clarity. Here we have used basic

camel-back notation to label each function in long form with a meaningful name. Moreover, the name clearly defines what the stated response means, so we need not resort to poring over the documentation every time we call a function. A conditional of the form,

```

    ifDoorSensorDoorClosed()then Dotheright
        thing
    else
        Makeafuss
    endif

```

is clear at a glance. *Functions* become *methods* in object-oriented languages, and will look a little different. Those who prefer typed variable-

naming conventions will create even longer names.
Regardless, the meaning should be transparent to you, to
your
code reviewer, and to whoever is gifted with the task of
maintaining your code two years from now.

www.newnespress.com

Each of these base-level routines can be tested in isolation to verify proper operation. For example:

```
PrimaryInterlockDoorTest() do
```

```
    IfPrimaryInterlockSensorDoorClosed()the (reportdoorclosed)
    else
        (reportdooropen)
    endif
loop
```

This trivial code allows basic testing to ensure that the door sensor is reading the door position correctly. As part of a verification plan, simple programs may be designed to exercise each low-level hardware routine while asking the tester to change the door state and record results. This auxiliary code is the equivalent of a test harness in the hardware world. Again, encapsulation lies at the heart of the “design-for-test” concept.

We can encapsulate the primary interlock system operation as follows:

```
PrimaryInterlockCheck()
```

```
    if PrimaryInterlockSensorDoorClosed() then switchOk
        1/4 SetPrimaryInterlockSwitch(true) ifswitchOk
        1/4 false then
            traphardwareerror
        endif
        switchOk 1/4 SetInterlockMonitorSwitch(false) ifswitchOk
        1/4 false then
            traphardwareerror
        endif
    else
        switchOk 1/4 SetPrimaryInterlockSwitch(false) ifswitchOk
        1/4 false then
            traphardwareerror
        endif
        switchOk 1/4 SetInterlockMonitorSwitch(true) ifswitchOk
        1/4 false then
            traphardwareerror
        endif
    endif
endif
```

The function returns true if the door interlock switch is properly closed, and false if it is not.

Physical interlock switches are set appropriately. If physical hardware fails to respond in a verifiable way,

the routine does not return at all; rather it executes special-purpose error handling code.

www.newnespress.com

The line “trap hardware error” is shorthand for abandoning ordinary operation in favor of a catchall operation that takes care of detectable hardware failures. This is sometimes called “throwing an exception,” developed further in discussion on “Exception Handling” later.

In a safety-critical system, this will generally result in an attempt to return everything as quickly as possible to a known safe state.

Every time this function is executed, the system tests the interlock sensor and sets the two control switches appropriate to the door state. Now we ask the question, how often do we execute this? Remember, the door itself is outside your control. Your practical options depend on your O/S and processor. Three options in decreasing order of desirability are considered:

Interrupt Driven: This critical operation is an obvious candidate for a hardware interrupt. If the primary interlock door sensor–state change can trigger a hardware interrupt, then `PrimaryInterlockCheck()` can be executed at each interrupt to change the relay states.

Schedule Driven: If the O/S supports multithread or scheduled execution, then `PrimaryInterlockCheck()` can be executed periodically on an independent thread from the primary operations software. “Periodically” must mean frequently enough to provide protection; hence, in this case, it will likely be on the order of no more than a few milliseconds between loops.

Main Loop: In very basic microprocessors, for simple tasks, one often relies on a tight main loop that tracks and triggers other operations. In this case, we would include the code:

```
do
    PrimaryInterlockCheck() querykeypad
    .. .
    dootherstuff
    .. .
loopuntilshutdown
```

The third method is common in “roll-your-own” real-time designs. We consider this common design pattern further in [section 5](#). It is risky, in that any glitch or bug in the “otherstuff” could freeze or delay the main loop, causing a safety hazard. This choice should be coupled with very careful version control, since any modification to the code that results in a new bug could create a severe safety hazard.

If forced to use method three, one should immediately

consider other places to add redundancy. For example, one presumes that somewhere in “otherstuff,” there will appear

www.newnespress.com

commands to turn on the magnetron; if available code space allows an extra call to

PrimaryInterlockCheck(), why not be safe?

Independence is key: *Safety critical functions should be designed, tested, and executed independently of control operations code.*

Methods 1 and 2 have the advantage, in that they will better survive code changes and new bugs in the operational code. They are by no means immune from the software failure modes and should also include strategic redundancy. However, with method 3, we are simulating periodic interrupts; hence adding redundant checks at other frequented portions of the control code should be considered, particularly where you expect longer latency. Function over form in this case, as safety is never inelegant. For a personal example, see [Example 2.4](#).

Keep in mind that this is a thought exercise and there is a fourth method, illustrated by the original schematic diagram: analog hardware. No software engineer should consider his or her system beyond fallibility and, in practice, one would not recommend implementing this or any safety-critical system without hardware interlock redundancy. Every critical-system software engineer should study the Therac-25 radiation therapy device failure (see [Example 2.8](#)) as inspiration.

Moving down the safety ladder, let us consider the secondary interlock system:

SecondaryInterlockCheck()

```

ifDoorSensorDoorClosed()then
    switchOk ¼ SetSecondaryInterlockSwitch(true) ifswitchOk
              ¼ setthen
    traphardwareerror
endif
else
    switchOk ¼ SetSecondaryInterlockSwitch(false) ifswitchOk
              ¼ setthen
    traphardwareerror
endif
endif
endif

```

As with the first interlock check, it makes sense to run this secondary code frequently in an interrupt-driven fashion or a scheduled thread. If we do not have that option, then, again, redundancy doesn't hurt. For example, we might write simple code to start cooking as follows:

TurnOnMagnetron()

PrimaryInterlockCheck()
SecondaryInterlockCheck()

www.newnespress.com

```

ifDoorSensorDoorClosed()then
    SwitchOK ¼ SetMagnetronEnergizeSwitch(true) ifswitchOk
    falsethen
        traphardwareerror
    endif
    magnetronActive ¼ true
else
    magnetronActive ¼ false
endif

```

As used here, `magnetronActive` is a state variable. This could correspond to a global variable in function-based code, or a property of the Magnetron class object in an OO language like Cþþ. It provides convenience for our control code, allowing us to determine the state where we think that we've placed the Magnetron. In fact, using the principles we have established, it would be better to make `magnetronActive` a function that actually tests the state of the magnetron. In this fashion, there is no way we could mislead ourselves into believing that the magnetron is energized when it is off, or (worse!) that it is off when in fact it is energized. Assuming that our hardware allows us to read back the register that controls relay 2, we can improve our design.

```

MagnetronActive()
    if(testmagnetronswitchstate) ¼ (closed)then
        returntrue
    else
        returnfalse
    endif

```

Even better would be to get independent secondary information, such as a direct measurement of current to the magnetron or direct detection of microwaves.

Example 2.8. The Therac-25 Nightmare

The Therac-25 radiation therapy debacle has become a standard case study in failure modes of software-based safety-critical systems. Between 1985 and 1987, multiple patients were given radiation doses roughly 100 times higher than intended as a result of an undiscovered software bug. At least six of these overdoses are known to have resulted in severe radiation poisoning and three of the patients eventually died.

This therapy system was feature-rich, with multiple distinct operating modes. In particular, the machine was able to deliver either short-duration direct electron-beam therapy or diffuse x-ray therapy. In order to produce x-rays, the system used the same electron beam at high energy directed against an emitting target. Radiation therapy devices represent an enormous capital investment and this was a clever engineering idea to maximize available treatment options

within a single unit. Secondary mechanisms were used to diffuse and collimate the beam to achieve properly controlled dosing for each mode.

The two operation modes required that diverse components be physically moved into and out of the beam path by mechanical actuation. Open-loop control was used to configure the mechanical components. Eventually, a hidden software bug emerged: when certain atypical user-interface entry sequences were made, the unit was temporarily misconfigured, sometimes resulting in a high power beam directed at the patient without being diffused to safe levels.

The Therac-25 software failed to verify hardware conditions. In fact, it has been reported that the open-loop control code had been retained from previous system designs in which hardware interlocks prevented such malfunctions; however, there were no hardware interlocks in the Therac-25. The problem triggered large-scale rethinking of the role of software and software process in safety-critical systems.

Not only was code inappropriately reapplied, the overall architecture was glued together in a way that made contextual verification very difficult. The designers probably relied on past field experience, unaware of having inherited hidden critical dependence on the hardware interlocks.

A chain of design failures contributed to the unsafe system, a list of which reads like a textbook nightmare example. Convenient features were prioritized compared to hazard analysis. The software was not independently reviewed. Borrowed code was reapplied without risk assessment. The hidden bug was triggered by rare keyboard entries, making detection in V&V unlikely and discovery and tracing in the field difficult. The machine detected irregularities in beam energy and shut down, but did so too slowly to protect the patient. Upon shutdown, only obscure alert codes were presented to the operator, offering no actionable information and encouraging complacency.

Each design failure compounded the others and, in not following good practices, the resulting system was not failsafe, user tolerant, nor engineering-flaw robust.

Numerous detailed reviews of the Therac-25 experience have been published, and we strongly encourage the reader to locate a few of them to help drive home the importance of

good process and design strategies. A convenient summary with a list of references is available online at en.wikipedia.org/wiki/Therac-25 and en.wikipedia.org/wiki/List_of_software_bugs.

Verify reality: Avoid mirroring hardware in state variables when the state is directly testable. In some cases you are at the mercy of the hardware or the physical system. The goal of the mission-critical software engineer, however, should be to ascertain and verify the actual state of the physical hardware insofar as possible so that critical decisions are not made

on faulty assumptions.

Following similar patterns, we can write straightforward code to test thermal conditions. For example,

```
MagnetronTempCheck()
    ifGetMagnetronTemperature(>135)then
        SwitchOK ¼ SetMagnetronThermalSwitch(false) ifswitchOk
        false then
            traphardwareerror
        endif returnfalse
    else
        SwitchOK ¼ SetMagnetronThermalSwitch(true) ifswitchOk
        false then
            traphardwareerror
        endif returntrue
    endif
```

Similarly, we can create a CavityTempCheck() function.

Leaving power surges and shorts to the good old hardware fuse, we now have four scheduled tasks to keep track of our safety state. We can bundle them for convenience:

```
SafetyStateCheck()
    PrimaryInterlockCheck() SecondaryInterlockCheck()
    MagnetronTempCheck() CavityTempCheck()
```

Finally, we can provide an encapsulated crosscheck routine that quickly verifies the state of all hardware and sensors so that the operational code can remain coupled to the state of the hardware and does not rely entirely on state variables.

```
SafeToOperate()
    If      PrimaryInterlockSensorDoorClosed()      and
            DoorSensorDoorClosed()                  and
            PrimaryInterlockRelayClosed()            and
            SecondaryInterlockRelayClosed()          and
            GetMagnetronTemperature() < 135          and
            GetCavityTemperature() < 120              then
                returntrue
    else
                returnfalse
    endif
```

Note what we have achieved. We have clearly named functions that perform our safety checks, bundled outside of the main control code so that they can be individually tested and readily verified by visual inspection in code review. Requirements generated by the hazard analysis are directly traceable to code.

Each function takes care of ensuring safety internally. Certain safety checks may return a status flag to the calling routine for purposes of notifying operating software that it is okay to continue to interact with the user. However, the actual safety of the system is not jeopardized even if these flags are ignored.

Now, with our safety-critical elements bundled, we can turn to implementing the control code. To keep things simple, let us assume this microwave oven has only one button: 60-second heat. Two mission-critical requirements emerge, one of which overlaps with our safety checks. These are:

1. Cook as close as possible to 60 seconds.
2. Interrupt if the door is opened.

Let us assume for maximum programmer hardship that we do not have interrupt or thread-based O/S support. We might code this sequentially as follows for our cooking checks first:

CookingChecks()

```

timerCnt    ¼ timerCnt—1 if
timerCnt    0 then
    TurnOffMagnetron()
    ifMagnetronActivethen PanicOff()
endif
endif

ifDoorSensorDoorClosed()    ¼ false then
    TurnOffMagnetron()
    ifMagnetronActivethen PanicOff()
endif
endif

```

Note we have put a direct crosscheck on the state of the magnetron. If it fails to go off when requested, we execute a panic routine that attempts to force the system to a known safe state. This is discussed further when we get to exception handling.

Next we code our tiny user interface:

```
ButtonCheck()
    if (button pushed) then SoundKeyPressBeep()
        TimerCnt      60(timefactor)
        SafetyStateCheck()

        IfSafeToOperate()then TurnOnMagnetron()

    else
        (complaintouser)
    endif
endif
```

“Complain to the user” in a simple system could be just a feedback beep that differs from the key-press sound, or could be a flashing error message. If the hardware allows, then longer explanations are more desirable. However, in the event of a dangerous condition out of keeping with expected events—such as a stuck relay or sensor—the designer should seriously consider disabling the interface in a manner that requires technician intervention. This would prevent a frustrated user from ignoring the alarm and possibly escalating the alarm condition into a situation that compromises safety.

Our main operation code is fairly simple:

```
Main()
    If SelfTestEmbeddedCodeOK()      ¼ false then
        (complaintouser)

        endexecution

    endif

    If SelfTestHardwareOK()          ¼ false then
        (complaintouser)

        endexecution

    endif SetInitialHardwareState() do
        SafetyStateCheck()

        ifMagnetronActivethen

            IfSafeToOperate()then CookingChecks()

            else
                PanicOff()
            endif

        else
```

```
ifDoorSensorDoorClosed()then CavityLightOff()  
    ButtonCheck()
```

www.newnespress.com

```

        else
            CavityLightOn()
        endif
    loop
endif

```

On cold re-start, the system first established that the software is uncompromised and that each testable hardware component and register responds properly—taking care, of course, not to exercise components during self-tests in any way that could itself compromise safety. Once we verify the system, we set initial hardware states and enter the main execution loop. Since this is an always-ready cooking appliance, there is no need for an explicit shut-down. That will occur either when the plug is pulled or an exception is detected.

We began with the safety-critical elements and then constructed the operational code. Note that we reused hardware status tests both in the safety-critical code and in the operational segments, simplifying implementation. We also make safety check calls at each level of the code, sometimes redundantly in the sense that we make similar checks at different code levels that might, when unwound, result in duplication in the sequence of events.

However, the duplication involves only the re-execution of tiny code segments and will not detract noticeably from response feel. Advantages include easy review of the code, since safety steps are visible at a glance at each level. Failure modes in our RA/HA chart can be mapped directly to independent, encapsulated code during critical safety reviews, while operational code can be verified against design operating sequences without nervously working through possible safety compromises at every software update. The redundancy also provides protection against future bugs introduced in the operational code and even some parts of the critical code segments.

Finally, a note on exception handling: In aviation, pilots have a general mnemonic phrase to keep priorities straight during emergencies. The phrase is, “Aviate, navigate, communicate.” The same principles apply to an emergency situation in a critical-system design.

When something goes wrong, priority one is to get the operation of the system under control and into a known safe state. The first step in this example would be to execute a `PanicOff()` operation. In practice, the difference between ordinary “off” routines and the panic routine comes down to assumptions about hardware compliance. The detected

failure could represent a transient symptom of an intermittent hardware issue. Rather than simply shutting down the magnetron by the usual means, it is advisable to issue API commands multiple times to each available interlock switch. Do so in a loop that tests for compliance and tests sensors for verification.

www.newnespress.com

Once the hardware is safe, then attempt to sort out the device status and whether it is safe to do anything other than remain shut down. Next, communicate the situation to the user.

If the shut-down loop times out without reaching a verified safe state, take all reasonable measures to otherwise shut down the system. Carefully consider this use

case during design of the safety-critical elements: How long should you wait to notify the user? How can you notify the user clearly and effectively to achieve desired action?

Generally your software can act far faster than any user can react; however, there are circumstances, such as precautionary personnel evacuation, where it will be advisable to sound the alarm in advance of trying to solve the problem and then communicate details later. Obviously, if active user intervention is required, notification then becomes the safety shut-down step.

Again, this thought exercise is not intended to show useful design patterns. It is not intended to be realistic, as one would never produce a microwave oven or any other dangerous physical device without including additional layers of hardware interlocks. Our purpose is to try to connect safety-first, failsafe redundant hardware engineering practices to similar possible practices in software.

Example 2.9. Telephone Tag

What about that perpetual “new call” light in my kitchen? I’ve never seen the firmware for my wall phone, but I’d wager a week of Venté Mochas that the error is conceptually similar to that described in [Example 2.4](#). A state variable of some sort (perhaps simply the register that controls the light) is intended to mirror reality. When it gets out of synch with reality, there is no periodic verification, and hence, in contrast to the brain stimulator, there is no automatic means of correcting the problem. Fortunately, this is not a mission-critical function of the phone; it is merely annoying.

A few years of sporadic test data reveal that the stuck “new call” light often occurs when the phone ring is interrupted before an incoming caller ID code is fully processed.

Let us fix these ideas with an example. Consider a block of code triggered by the ringing phone:

```
OnRing()
```

```
...
```

```
setNewCallLight
```

```
newADisplayThread.start
```

and somewhere in the GetCallerID thread code:

```
getInComingCallerID()
...
dountillastbyte received getdata
    accumulate CallIDstring if
    waitTimeOutthen
        return "NoCallerIDAvailable" endif
...
loop
...
addCallerIDtoCallList(CallIDstring) newCallCount ¼
newCallCount þ 1 return CallIDstring
```

Most of the time the phone will ring and a fully processed result will be added to the call list. On this assumption, the engineer probably delegates clearing of the “new call” light to the call review operation: each time a call is viewed in the list, the individual “new” flag is cleared and the newCallCount is decremented. If newCallCount hits zero, then the clearNewCallLight is executed. However, if the newCallCount ¼ 0 already, then clearNewCallLight is never executed. A classic coding error is to use too specific a conditional:

```
...
newCallCount ¼ newCallCount—1 if
newCallCount ¼ 0 then
    clearNewCallLight endif
...
```

The problem, of course, is that if newCallCount ¼ 0 already, then the decrement (depending upon the arithmetic type) either takes the counter negative or wraps to the highest possible integer. With all individual “new” flags cleared, the counter is now misaligned with the actual number of new calls, and the counter light will likely never be cleared no matter how many times we cycle through the review list.

How could this happen? Caller ID is generally transmitted over the beginning of the first ring period; if it is possible to pick up the phone or otherwise interrupt the process between turning on the “new call” light and incrementing the newCallCount, then we will have triggered the intermittent bug.

This is only one concocted scenario and there are ample other possibilities. Perhaps the “new call” light-circuit switch simply intermittently fails to respond and the code never revisits turning off the light once the internal state reflects an off condition. We can be sure that it is not a pure hardware error, such as a stuck gate, since clearing all calls from memory with a “soft reset” reliably switches off the light.

(Continues)

Example 2.9. Cont'd

It does not so much matter what is failing—an interrupted firmware thread, or a slightly sticky bit. Ultimately, the engineer has made an assumption that the state of the light will always remain in synch and has failed to subsequently verify reality in the code.

Defensive programming possibilities abound: forcing the indicator light and program state to synchronize at any regular point in the phone operation could have easily provided a catchall check. Certainly the user expects the new call light to go out when all calls are cleared. Why not make a quick count of “new” call flags in the list as the software is leaving “call review” mode, and simply force the light “off” whenever that count comes up zero? Perhaps that is too time consuming for the tiny embedded processor; then why not at least force the light “off” when the user scrolls back to call number #1 in the review list? If the user has reviewed all the calls in the list, then he or she has obviously seen them all.

This is a benign example of where good coding practices could improve the user experience. Obviously, our curmudgeonly light is not safety sensitive, nor is it mission critical to the operation of the telephone. But it certainly was not the intent of the architect to force phone users to exercise the soft reset—deleting all incoming call records—on a regular basis. Lack of redundancy and lack of provability led to a low-occurrence failure mode that escaped testing. By providing independent secondary code elements as a matter of course one can easily design in low-overhead redundancy that ensures a reliable user experience.

7. Conclusions

We discussed three principles that drive all aspects of the critical-system development process that persist through design, implementation, and testing. Our ultimate goal is the last of these: provability. To achieve it we need to carefully apply the other two, and broaden our view with respect to how we think about failure modes, how we think about the fallibility of our system components, and how we think about the fallibility of human designers.

The second principle, redundancy, like the second law of thermodynamics, is perhaps the most significant in obtaining useful results. Strategic redundancy at all stages of the design process and in the code architecture underlies consistently successful critical-system development.

Critical-safety functions should be designed, tested, encapsulated, and executed independently of control operations code insofar as possible. Design these elements first, begin with a carefully thought-out RA/HA matrix that includes hardware, environmental, software, and user-associated risks. Verify the critical backbone subsystems independently before focusing on integration with operational code. To the extent possible, meet specific needs with specific, traceable code. Think of it as defensive programming.

Where possible, always design systems to verify reality: avoid open-loop designs and mirroring hardware with state variables when the physical state is directly measurable. Consider the user and the user interface in terms of how they affect the mission risks and establish explicit use cases; make sure that information conveyed is necessary, clear, and actionable and that feature creep does not overrun function.

Even non-critical embedded systems can benefit from some of these design patterns.

All embedded software has a mission; predictably, achieving that mission is what makes a good-quality product. (My kitchen phone still insists I have a new call.)

Failsafe software design means acknowledging a broad spectrum of downside threats and possibilities and carefully bounding the risks. No human engineered system is inherently infallible and we should not design systems on the expectation that we can make them so. Instead we can engineer for anticipated failure modes insofar as possible and for unanticipated failure modes through built-in redundancy and safety factors. By taking considered steps and exercising good practices, we can create critical systems that are robust to physical component failures and unplanned situations, highly user tolerant, and even forgiving of the system's own potential hidden flaws.

3. Distributed Control of Robotic Network

Chapter One

An introduction to distributed algorithms

Graph theory, distributed algorithms, and linear distributed algorithms are a fascinating scientific subject. In this chapter we provide a broad introduction to distributed algorithms by reviewing some preliminary graphical concepts and by studying some simple algorithms. We begin the chapter with one section introducing some basic notation and another section stating a few useful facts from matrix theory, dynamical systems, and convergence theorems based on invariance principles. In the third section of the chapter, we provide a primer on graph theory with a particular emphasis on algebraic aspects, such as the properties of adjacency and Laplacian matrices associated to a weighted digraph. In the next section of the chapter, we introduce the notion of synchronous network and of distributed algorithm. We state various complexity notions and study them in simple example problems such as the broadcast problem, the tree computation problem, and the leader election problem. In the fifth section of the chapter, we discuss linear distributed algorithms. We focus on linear algorithms defined by sequences of stochastic matrices and review the results on their convergence properties. We end the chapter with three sections on, respectively, bibliographical notes, proofs of the results presented in the chapter, and exercises.

1.1 ELEMENTARY CONCEPTS AND NOTATION

1.1.1 Sets and maps

We assume that the reader is familiar with basic notions from topology, such as the notions of open, closed, bounded, and compact sets. In this section, we just introduce some basic notation. We let $x \in S$ denote a point x belonging to a set S . If S is finite, we let $|S|$ denote the number of its elements. For a set S , we let $P(S)$ and $F(S)$

denote the collection of subsets of S and the collection of finite subsets of S , respectively. The empty set is denoted by $\emptyset$. The interior and the boundary of a set S are denoted by $\text{int}(S)$ and ∂S , respectively. If R is a subset of or equal to S , then we write $R \subset S$. If R is a strict subset of S , then we write $R \subsetneq S$. We describe

subsets of S defined by specific conditions via the notation

$$\{x \in S \mid \text{condition(s) on } x\}.$$

Given two sets S_1 and S_2 , we let $S_1 \cup S_2$, $S_1 \cap S_2$, and $S_1 \times S_2$ denote the union, intersection, and Cartesian product of S_1 and S_2 , respectively. Given a collection of sets $\{S_a\}_{a \in A}$ indexed by a set A , we interchangeably denote their Cartesian product by $\prod_{a \in A} S_a$ or by $S_a \mid a \in A$. We

adopt analogous notations for union and intersection. We denote by S^n the Cartesian product of n copies of the same S . The *diagonal set* $\text{diag}(S^n)$ of S^n is given by $\text{diag}(S^n) = \{(s, \dots, s) \in S^n \mid s \in S\}$. The set $S_1 \setminus S_2$ contains all points in S_1 that do not belong to S_2 .

We let $\mathbb{N}$ and $\mathbb{Z}_{\geq 0}$ denote the set of natural numbers and of non-negative integers, respectively. We let $\mathbb{R}$, $\mathbb{R}_{>0}$, $\mathbb{R}_{\geq 0}$, and $\mathbb{C}$ denote the set of real numbers, strictly positive real numbers, non-negative real numbers, and complex numbers, respectively. The sets $\mathbb{R}^d$, $\mathbb{C}^d$, and $S^d \subset \mathbb{R}^{d+1}$ are the d -dimensional Euclidean space, the d -dimensional complex space, and the d -dimensional sphere, respectively. The tangent space of $\mathbb{R}^d$, denoted by $T\mathbb{R}^d$, is the set of all vectors tangent to $\mathbb{R}^d$. Note that $T\mathbb{R}^d$ can be identified with $\mathbb{R}^d$ by mapping a vector v tangent to $\mathbb{R}^d$ at $x \in \mathbb{R}^d$ to the pair (x, v) . Likewise, $T S^d$ is the set of all vectors tangent to S^d , and can be identified with $S^d \times \mathbb{R}^d$. The Euclidean space $\mathbb{R}^d$ contains the vectors $0_d = (0, \dots, 0)$, $1_d = (1, \dots, 1)$, and the standard basis $e_1 = (1, 0, \dots, 0), \dots, e_d = (0, \dots, 0, 1)$. Given $a < b$, we let $[a, b]$ and $]a, b[$ denote the closed interval and the open

interval between a and b , respectively.

Given two sets S and T , we let $f : S \rightarrow T$ denote a map from S to T , that is, a unique way of associating an element of T to an element of S .

The *image* of the map $f : S \rightarrow T$ is the set $\text{image}(f) = \{f(s) \in T \mid s \in S\}$. Given the map $f : S \rightarrow T$ and a set $S_1 \subset S$, we let $f(S_1) = \{f(s) \mid s \in S_1\}$ denote the image of the set S_1 under the map f . Given $f : S \rightarrow T$ and $g : U \rightarrow S$, we let $f \circ g : U \rightarrow T$, defined by $f \circ g(u) = f(g(u))$, denote the composition of f and g . The map $\text{id}_S : S \rightarrow S$ is the identity map on

S . Given $f: S \rightarrow \mathbb{R}$, the *support* of f is the set of elements s such that $f(s) \neq 0$. Given a subset $R \subseteq S$, the indicator map $1_R: S \rightarrow \mathbb{R}$ associated with R is given by $1_R(q) = 1$ if $q \in R$, and $1_R(q) = 0$ if $q \notin R$. Given two sets S and T , a *set-valued map*, denoted by $h: S \rightrightarrows T$, associates to each element of S a subset of T . Given a map $f: S \rightarrow T$, the *inverse map* $f^{-1}: T \rightrightarrows S$ is defined by

$$f^{-1}(t) = \{s \in S \mid f(s) = t\}.$$

If f is a real-valued function, that is, a function of the form $f: S \rightarrow \mathbb{R}$, then $f^{-1}(x) \subseteq S$, for any $x \in \mathbb{R}$, is a *level set* of f . In what follows, we require the reader to be familiar with some basic smoothness notions

for functions. Specifically, we will use the notions of locally and globally Lipschitz functions, differentiable, piecewise differentiable and continuously differentiable functions, and functions that are multiple times differentiable.

Finally, we introduce the so-called *Bachmann–Landau symbols*. For $f, g: \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, we say that $f \in O(g)$ (resp., $f \in \Omega(g)$) if there exist $n_0 \in \mathbb{N}$ and $K \in \mathbb{R}_{>0}$ (resp., $k \in \mathbb{R}_{>0}$) such that $f(n) \leq Kg(n)$ for all $n \geq n_0$ (resp., $f(n) \geq kg(n)$ for all $n \geq n_0$). If $f \in O(g)$ and $f \in \Omega(g)$, then we use the notation $f \in \Theta(g)$.

1.1.2 Distance functions

A function $\text{dist}: S \times S \rightarrow \mathbb{R}_{\geq 0}$ defines a *distance* on a set S if it satisfies:

(i) $\text{dist}(x, y) = 0$ if and only if $x = y$; (ii) $\text{dist}(x, y) = \text{dist}(y, x)$, for all $x, y \in S$; and (iii) $\text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y)$, for all $x, y, z \in S$. The pair (S, dist) is usually called a *metric space*.

Some relevant examples of distance functions include the following:

L^p -distance on $\mathbb{R}^d$. For $p \in [1, +\infty[$, consider the L^p -norm on $\mathbb{R}^d$ defined by $\|x\|_p = \left(\sum_{i=1}^d |x_i|^p \right)^{1/p}$. For $p = +\infty$, consider the L^∞ -norm on $\mathbb{R}^d$ defined by $\|x\|_\infty = \max_{i \in \{1, \dots, d\}} |x_i|$. Any of these norms defines

naturally a L^p -distance in $\mathbb{R}^d$ by $\text{dist}_p(x, y) = \|y - x\|_p$. In particular, the most widely used is the Euclidean distance, corresponding to $p = 2$. Unless otherwise noted, it is always understood that $\mathbb{R}^d$ is endowed with this notion of distance. We will also use the L^1 - and the L^∞ -distances. Finally, it is convenient to define the norm $\|z\|_1$ of

a complex number $z \in \mathbb{C}$ to be the Euclidean norm of z regarded as a vector in $\mathbb{R}^2$.

Geodesic distance on S^d . Another example is the notion of *geodesic distance* on S^d . This is defined as follows. For $x, y \in S^d$, $\text{dist}_g(x, y)$ is the length of the shortest curve in S^d connecting x and y . We will use this notion of distance in dimensions $d = 1$ and $d = 2$. On the unit circle S^1 , by convention, let us define positions as angles measured counterclockwise from the positive horizontal axis. Then, the geodesic distance can be expressed as

$$\text{dist}_g(x, y) = \min\{\text{dist}_c(x, y), \text{dist}_{cc}(x, y)\}, \quad x, y \in S^1,$$

where $\text{dist}_c(x, y) = (x - y) \bmod 2\pi$ is the clockwise distance and $\text{dist}_{cc}(x, y) = (y - x) \bmod 2\pi$ is the counterclockwise distance. Here the clockwise distance between two angles is the path length from an angle to the other traveling clockwise, and $x \bmod 2\pi$ is the remainder

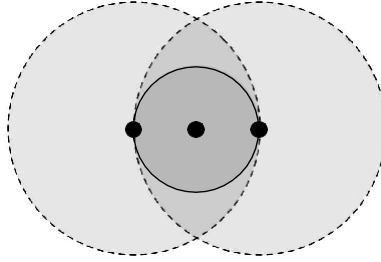


Figure 1.1 Open balls (dashed lines), a closed ball (solid line), and an open lune for the Euclidean distance on the plane.

of the division of x by 2π . On the sphere S^2 , the geodesic distance can be computed as follows. Given $x, y \in S^2$, one considers the great circle determined by x and y . Then, the geodesic distance between x and y is exactly the length of the shortest arc in the great circle connecting x and y .

Cartesian product distance on $\mathbb{R}^{d_1} \times S^{d_2}$. Consider $\mathbb{R}^{d_1}$ endowed with an L^p -distance, $p \in [1, +\infty]$, and S^{d_2} endowed with the geodesic distance. Given $(x_1, y_1), (x_2, y_2) \in \mathbb{R}^{d_1} \times S^{d_2}$, their *Cartesian product distance* is given by $\text{dist}_p(x_1, x_2) + \text{dist}_g(y_1, y_2)$. Unless otherwise noted, it is understood that $\mathbb{R}^{d_1} \times S^{d_2}$ is endowed with the Cartesian product distance $(\text{dist}_2, \text{dist}_g)$.

Given a metric space (S, dist) , the *open* and *closed balls* of center $x \in S$ and radius $\varepsilon \in \mathbb{R}_{>0}$ are defined by, respectively,

$$B(x, \varepsilon) = \{y \in S \mid \text{dist}(x, y) < \varepsilon\},$$

$$\overline{B}(x, \varepsilon) = \{y \in S \mid \text{dist}(x, y) \leq \varepsilon\}.$$

Consider a point $x \in X$ and a set $S \subseteq X$. A *neighborhood* of a point $x \in X$ is a subset of X that contains an open ball centered at x . A *neighborhood* of a set $Y \subseteq X$ is a subset of X that, for each point $y \in Y$, contains an open ball centered at y . The *open lune* associated to $x, y \in S$ is $B(x, \text{dist}(x, y)) \setminus B(y, \text{dist}(x, y))$. These notions are illustrated in Figure 1.1 for the plane equipped with the Euclidean distance.

Given a metric space (S, dist) , the distance between a

point $x \in S$ and a set $W \subset S$ is the infimum of all distances between x and each of the points in W . Formally, we set

$$\text{dist}(x, W) = \inf\{\text{dist}(x, y) \mid y \in W\}.$$

The projection of a point $x \in S$ onto a set $W \subset S$ is the set-valued map

4

$\text{proj}_W : S \rightarrow W$ defined by

$$\text{proj}_W(x) = \{y \in W \mid \text{dist}(x, y) = \text{dist}(x, W)\}.$$

If W is a closed set, then $\text{proj}_W(x) \neq \emptyset$ for any $x \in S$. The *diameter* of a set is the maximum distance between any two points in the set; formally, we set $\text{diam}(S) = \sup\{\text{dist}(x, y) \mid x, y \in S\}$. With a slight abuse of notation, we often use $\text{diam}(P)$ to denote $\text{diam}(\{p_1, \dots, p_n\})$ for $P = (p_1, \dots, p_n)$.

1.1.3 Curves

A curve is the image of a continuous map $\gamma : [a, b] \rightarrow \mathbb{R}^d$. The map γ is called a *parameterization* of the curve. We usually identify a parameterization with the curve it defines. Without loss of generality, any curve can be given a parametrization with $a = 0$ and $b = 1$. A curve connects the two points p and q if $\gamma(0) = p$ and $\gamma(1) = q$. A curve $\gamma : [0, 1] \rightarrow \mathbb{R}^d$ is *not self-intersecting* if γ is injective on $(0, 1)$. A curve is *closed* if $\gamma(0) = \gamma(1)$.

A set $S \subset \mathbb{R}^d$ is *path connected* if any two points in S can be joined by a curve. A set $S \subset X$ is *simply connected* if it is path connected and any

not self-intersecting closed curve can be continuously deformed to a point in the set; that is, for any injective continuous map $\gamma : [0, 1] \rightarrow S$ that satisfies $\gamma(0) = \gamma(1)$, there exist $p \in S$ and a continuous map $H : [0, 1] \times [0, 1] \rightarrow S$ such that $H(t, 0) = \gamma(t)$ and $H(t, 1) = p$ for all $t \in [0, 1]$. Informally, a simply connected set is a set that consists of a single piece and does not have any holes.

Next, consider a piecewise continuously differentiable curve $\gamma : [0, 1] \rightarrow \mathbb{R}^d$;

the *length* of γ is

$$\text{length}(\gamma) = \int_0^1 \|\dot{\gamma}(s)\|_2 ds,$$

and its *arc-length parameter* is

$$s = \int_0^s \|\dot{\gamma}(t)\|_2 dt.$$

$$s_{\text{arc}}(s) =$$

$$0$$

Note that as the parameter t varies in $[0, 1]$, the arc-length parameter $s_{\text{arc}}(t)$ varies in $[0, \text{length}(\gamma)]$. The *arc-length parameterization* of the curve is the map $\gamma_{\text{arc}} : [0, \text{length}(\gamma)] \rightarrow \mathbb{R}^d$ defined by the equation $\gamma_{\text{arc}}(s_{\text{arc}}(s)) = \gamma(s)$.

With a slight abuse of notation, we will often drop the subindex *arc* and denote the arc-length parameterization by γ too.

For closed, not self-intersecting curves in the plane, we introduce the notion of signed and absolute curvatures as follows. Let $\gamma : [0, \text{length}(\gamma)] \rightarrow$

$\mathbb{R}^2$ be the counterclockwise arc-length parameterization of a curve. Assume γ is closed, not self-intersecting and twice continuously differentiable. Define

the *tangent vector* $\gamma' : [0, \text{length}(\gamma)] \rightarrow \mathbb{R}^2$ by $\gamma'(s) = \frac{d\gamma}{ds}$. Note that the

tangent vector has unit length, that is, $\|\gamma'(s)\|_2 = 1$ for all s . Additionally, define the *outward normal vector* $n_{\text{out}} : [0, \text{length}(\gamma)] \rightarrow \mathbb{R}^2$ to be the unit-length vector that is point-wise orthogonal to the tangent vector and directed outside the set enclosed by the closed curve γ . With these notations, the *signed curvature* $\kappa_{\text{signed}} : [0, \text{length}(\gamma)] \rightarrow \mathbb{R}$ is defined by requiring that it satisfies

$$\gamma''(s) = -\kappa_{\text{signed}}(s) n_{\text{out}}(s), \quad \text{and} \quad n'_{\text{out}}(s) = \kappa_{\text{signed}}(s) \gamma'(s).$$

If the set enclosed by the closed curve γ is strictly convex, then the signed curvature of γ is strictly positive. In general, the (*absolute*) *curvature* $\kappa_{\text{abs}} : [0, \text{length}(\gamma)] \rightarrow \mathbb{R}_{\geq 0}$ and the *radius of curvature* $\rho : [0, \text{length}(\gamma)] \rightarrow \mathbb{R}_{\geq 0}$ of the curve γ are defined by, respectively,

$$\kappa_{\text{abs}}(s) = |\kappa_{\text{signed}}(s)|, \quad \text{and} \quad \rho(s) = |\kappa_{\text{signed}}(s)|^{-1}.$$

1.2 MATRIX THEORY

Here, we present basic notions and results about matrix theory, following the treatments in [Horn and Johnson \(1985\)](#) and [Meyer \(2001\)](#). We let $\mathbb{R}^{n \times m}$ and $\mathbb{C}^{n \times m}$ denote the set of $n \times m$ real and complex matrices. Given a real matrix

A and a complex matrix U , we let A^T and U^* denote the transpose of A and the conjugate transpose matrix of U , respectively. We let I_n denote the $n \times n$ identity matrix. For a square matrix A , we write $A > 0$, resp. $A \geq 0$, if A is symmetric positive definite, resp. symmetric positive semidefinite. For a real matrix A , we let $\text{kernel}(A)$ and $\text{rank}(A)$ denote the kernel and rank of A , respectively. Given a vector v , we let $\text{diag}(v)$ denote the square matrix whose diagonal elements are equal to the component v and whose off-diagonal elements are zero.

1.2.1 Matrix sets

A matrix $A \in \mathbb{R}^{n \times n}$ with entries a_{ij} , $i, j \in \{1, \dots, n\}$, is

- (i) *Orthogonal* if $AA^T = I_n$, and is *special orthogonal* if it is orthogonal with $\det(A) = +1$. The set of orthogonal matrices is a group.¹

¹A set tt with a binary operation, denoted by $tt \times tt \ni (a, b) \mapsto a \star b \in tt$, is a *group* if: (i) $a \star (b \star c) = (a \star b) \star c$ for all $a, b, c \in tt$ (associativity property); (ii) there exists $e \in tt$ such that $a \star e = e \star a = a$ for all $a \in tt$ (existence of an identity element); and (iii) there exists $a^{-1} \in tt$ such that $a \star a^{-1} = a^{-1} \star a = e$ for all $a \in tt$ (existence of inverse elements).

- (ii) *Nonnegative* (resp., *positive*) if all its entries are nonnegative (resp., positive).
- (iii) *Row-stochastic* (or *stochastic* for brevity) if it is nonnegative and $\sum_{j=1}^n a_{ij} = 1$, for all $i \in \{1, \dots, n\}$; in other words, A is row-stochastic if $Al_n = 1_n$.
- (iv) *Column-stochastic* if it is nonnegative and $\sum_{i=1}^n a_{ij} = 1$, for all $j \in \{1, \dots, n\}$.
- (v) *Doubly stochastic* if A is row-stochastic and column-stochastic.
- (vi) *Normal* if $A^T A = AA^T$.
- (vii) A *permutation matrix* if A has precisely one entry equal to one in each row, one entry equal to one in each column, and all other entries equal to zero. The set of permutation matrices is a group.

The scalars $\mu_1, \dots, \mu_k$ are *convex combination coefficients* if $\mu_i \geq 0$, for $i \in \{1, \dots, k\}$, and $\sum_{i=1}^k \mu_i = 1$. (Each row of a row-stochastic matrix contains convex combination coefficients.) A *convex combination* of vectors

is a linear combination of the vectors with convex combination coefficients. A subset U of a vector space V is *convex* if the convex combination of any two elements of U takes value in U . For example, the set of stochastic matrices and the set of doubly stochastic matrices are convex.

Theorem 1.1 (Birkhoff – von Neumann). *A square matrix is doubly stochastic if and only if it is a convex combination of permutation matrices.*

Next, we review two families of relevant matrices with useful properties. *Toeplitz matrices* are square matrices with equal entries along each diagonal parallel to the main diagonal. In other words, a Toeplitz matrix is a matrix of the form

$$\begin{bmatrix} t_0 & t_1 & \cdots & \cdots & \cdots & t_{n-2} & t_{n-1} \\ & \cdot & \cdot & \cdot & \cdot & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \cdot & \cdot & \cdot & \end{bmatrix}$$

$$\begin{array}{cccccccc}
\boxed{} & t_{-1} & t_0 & t_1 & \cdots & \cdot & \cdots & t_{n-2} \\
& & & & & \cdot & & \\
\boxed{} & \cdots & t_{\boxed{}} & t & t & \cdots & \cdots & \cdots \\
\boxed{} & \cdots & \cdots^{-1} & t^0 & t^{-1} & t & \cdots & \cdots \cdot \\
& \cdots & \cdots & \cdots^{-1} & t^0 & t^1 & t & \cdots \boxed{} \\
& & & -1 & 0 & 1 & & \\
& t & \cdots & \cdots & \cdots & t & t & t \\
\boxed{}^{-n+2} & & & & -1 & 0 & 1 & \boxed{} \\
& t_{-n+1} & t_{-n+2} & \cdots & \cdots & \cdots & t_{-1} & t_0
\end{array}$$

7

An $n \times n$ Toeplitz matrix is determined by its first row and column, and hence by $2n - 1$ scalars.

Circulant matrices are square Toeplitz matrices where each two subsequent row vectors v_i and v_{i+1} have the following two properties: the last entry of v_i is the first entry of v_{i+1} and the first $(n - 1)$ entries of v_i are the second $(n - 1)$ entries of v_{i+1} . In other words, a circulant matrix is a matrix of the form

$$\begin{bmatrix}
 c_0 & c_1 & \cdots & \cdots & \cdots & c_{n-2} & c_{n-1} \\
 & & & & & & \\
 & & & & & & \\
 & & & & & & \\
 c_{n-1} & c_0 & c_1 & \cdots & \cdots & \cdots & c_{n-2} \\
 & & & & & & \\
 & & & & & & \\
 \cdots & c & c & c & \cdots & \cdots & \cdots \\
 \cdots & \cdots & \cdots & c_{n-1}^0 & c_0^1 & c_1 & \cdots \\
 \cdots & \cdots & \cdots & \cdots & c_{n-1} & c_0 & c_1 \\
 c_2 & \cdots & \cdots & \cdots & \cdots & c_{n-1} & c_0 & c_1 \\
 & & & & & & & \\
 c_1 & c_2 & \cdots & \cdots & \cdots & c_{n-1} & c_0
 \end{bmatrix}$$

and, therefore, it is determined by its first row.

1.2.2 Eigenvalues, singular values, and induced norms

We require the reader to be familiar with the notion of eigenvalue and of simple eigenvalue, that is, an eigenvalue with algebraic and geometric multiplicity² equal to 1.

The set of eigenvalues of a matrix $A \in \mathbb{R}^{n \times n}$ is called its *spectrum* and is denoted by $\text{spec}(A) \subset \mathbb{C}$. The *singular values* of the matrix

$A \in \mathbb{R}^{n \times n}$ are the positive square roots of the eigenvalues of $A^T A$.

We begin with a well-known property of the spectrum of a matrix.

Theorem 1.2 (Geršgorin disks). *Let A be an $n \times n$ matrix. Then*

$$\text{spec}(A) \subset \bigcup_{i \in \{1, \dots, n\}} \left\{ z \in \mathbb{C} : \|z - a_{ii}\|_{\mathbb{C}} \leq \sum_{j=1, j \neq i}^n |a_{ij}| \right\}.$$

Next, we review a few facts about normal matrices, their eigenvectors and their singular values.

²The algebraic multiplicity of an eigenvalue is the multiplicity of the corresponding root of the characteristic equation. The geometric multiplicity of an eigenvalue is the number of linearly independent eigenvectors corresponding to the eigenvalue. The algebraic multiplicity is greater than or equal to the geometric multiplicity.

Lemma 1.3 (Normal matrices). *For a matrix $A \in \mathbb{R}^{n \times n}$, the following statements are equivalent:*

- (i) *A is normal;*
- (ii) *A has a complete orthonormal set of eigenvectors; and*
- (iii) *A is unitarily similar to a diagonal matrix, that is, there exists a unitary³ matrix U such that U^*AU is diagonal.*

Lemma 1.4 (Singular values of a normal matrix). *If a normal matrix has eigenvalues $\{\lambda_1, \dots, \lambda_n\}$, then its singular values are $\{|\lambda_1|, \dots, |\lambda_n|\}$.*

It is well known that real symmetric matrices are normal, are diagonalizable by orthogonal matrices, and have real eigenvalues. Additionally, circulant matrices are normal.

We conclude by defining the notion of induced norm of a matrix. For

$p \in \mathbb{N} \cup \{\infty\}$, the p -induced norm of $A \in \mathbb{R}^{n \times n}$ is

$$\|A\|_p = \max\{\|Ax\|_p \mid \|x\|_p = 1\}.$$

One can see that

$$\begin{aligned} \|A\|_1 &= \max_{j \in \{1, \dots, n\}} \sum_{i=1}^n |a_{ij}|, & \|A\|_\infty &= \max_{i \in \{1, \dots, n\}} \sum_{j=1}^n |a_{ij}|, \\ \|A\|_2 &= \max\{\sigma \mid \sigma \text{ is a singular value of } A\}. \end{aligned}$$

1.2.3 Spectral radius and convergent matrices

The *spectral radius* of a matrix $A \in \mathbb{R}^{n \times n}$ is

$$\rho(A) = \max\{|\lambda| \mid \lambda \in \text{spec}(A)\}.$$

In other words, $\rho(A)$ is the radius of the smallest disk centered at the origin that contains the spectrum of A.

Lemma 1.5 (Induced norms and spectral radius). *For any square matrix A and in any norm $p \in \mathbb{N} \cup \{\infty\}$, $\rho(A) \leq \|A\|_p$.*

We will often deal with matrices with an eigenvalue equal to 1 and all other eigenvalues strictly inside the unit

disk. Accordingly, we generalize the notion of spectral radius as follows. For a square matrix A with $\rho(A) = 1$, we define the *essential spectral radius*

$$\rho_{\text{ess}}(A) = \max\{|\lambda| \mid \lambda \in \text{spec}(A) \setminus \{1\}\}. \quad (1.2.1)$$

³A complex matrix $U \in \mathbb{C}^{n \times n}$ is *unitary* if $U^{-1} = U^*$.

Next, we will consider matrices with useful convergence properties.

Definition 1.6 (Convergent and semi-convergent matrices). A matrix $A \in \mathbf{R}^{n \times n}$ is

- (i) *semi-convergent* if $\lim_{l \rightarrow +\infty} A^l$ exists; and
- (ii) *convergent* if it is semi-convergent and $\lim_{l \rightarrow +\infty} A^l = 0$.

These two notions are characterized as follows.

Lemma 1.7 (Convergent and semi-convergent matrices). *The square matrix A is convergent if and only if $\rho(A) < 1$. Furthermore, A is semi-convergent if and only if the following three properties hold:*

- (i) $\rho(A) \leq 1$;
- (ii) $\rho_{\text{ess}}(A) < 1$, that is, 1 is an eigenvalue and 1 is the only eigenvalue on the unit circle; and
- (iii) the eigenvalue 1 is semisimple, that is, it has equal algebraic and geometric multiplicity (possibly larger than one).

In other words, A is semi-convergent if and only if there exists a nonsingular matrix T such that

$$A = T \begin{pmatrix} I_k & 0 \\ 0 & B \end{pmatrix} T^{-1},$$

where $B \in \mathbf{R}^{(n-k) \times (n-k)}$ is convergent, that is, $\rho(B) < 1$. With this notation, we have $\rho_{\text{ess}}(A) = \rho(B)$ and the algebraic and geometric multiplicity of the eigenvalue 1 is k .

1.2.4 Perron–Frobenius theory

Positive and nonnegative matrices have useful spectral properties. In what follows, the first theorem amounts to the original Perron’s Theorem for positive matrices and the following theorems are the extension due to

Frobenius for certain nonnegative matrices. We refer to (Horn and Johnson, 1985, Chapter 8) for a detailed treatment.

Theorem 1.8 (Perron-Frobenius for positive matrices). *If the square matrix A is positive, then*

(i) $\rho(A) > 0$;

- (ii) $\rho(A)$ is an eigenvalue, it is simple, and $\rho(A)$ is strictly larger than the magnitude of any other eigenvalue; and
- (iii) $\rho(A)$ has an eigenvector with positive components.

Requiring the matrix to be strictly positive is a key assumption that limits the applicability of this theorem. It turns out that it is possible to obtain the same results of the theorem under weaker assumptions.

Definition 1.9 (Irreducible matrix). A nonnegative matrix $A \in \mathbb{R}^{n \times n}$ is *irreducible* if, for any nontrivial partition $J \cup K$ of the index set $\{1, \dots, n\}$, there exist $j \in J$ and $k \in K$ such that $a_{jk} > 0$.

Remark 1.10 (Properties of irreducible matrices). An equivalent definition of irreducibility is given as follows. A matrix $A \in \mathbb{R}^{n \times n}$ is *irreducible* if it is not reducible, and is *reducible* if either:

- (i) $n = 1$ and $A = 0$; or
- (ii) there exists a permutation matrix $P \in \mathbb{R}^{n \times n}$ and a number $r \in \{1, \dots, n-1\}$ such that $P^T A P$ is block upper triangular with diagonal blocks of dimensions $r \times r$ and $(n-r) \times (n-r)$.

It is an immediate consequence that the property of irreducibility depends upon only the patterns of zeros and nonzero elements of the matrix. •

We can now weaken the assumption in Theorem 1.8 and obtain a comparable, but weaker, result for irreducible matrices.

Theorem 1.11 (Perron-Frobenius for irreducible matrices). *If the nonnegative square matrix A is irreducible, then*

- (i) $\rho(A) > 0$;
- (ii) $\rho(A)$ is an eigenvalue, and it is simple; and
- (iii) $\rho(A)$ has an eigenvector with positive components.

In general, the spectral radius of a nonnegative irreducible matrix does not need to be the only eigenvalue of maximum magnitude. For example, the

matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ has eigenvalues $-1, 1$. In other words, irreducible matrices $\begin{pmatrix} * & * \\ * & * \end{pmatrix}$ do indeed have weaker spectral properties than positive matrices.

Therefore, it remains unclear which nonnegative matrices have the same properties as those stated for positive matrices in Theorem 1.8.

Definition 1.12 (Primitive matrix). A nonnegative square matrix A is

primitive if there exists $k \in \mathbb{N}$ such that A^k is positive.

It is easy to see that if a nonnegative square matrix is primitive, then it is irreducible. In later sections we will provide a graph-theoretical characterization of primitive matrices; for now, we are finally in a position to sharpen the results of Theorem 1.11.

Theorem 1.13 (Perron-Frobenius for primitive matrices). *If the nonnegative square matrix A is primitive, then*

- (i) $\rho(A) > 0$;
- (ii) $\rho(A)$ is an eigenvalue, it is simple, and $\rho(A)$ is strictly larger than the magnitude of any other eigenvalue; and
- (iii) $\rho(A)$ has an eigenvector with positive components.

We conclude this section by noting the following convergence property that is an immediate corollary to Lemma 1.7 and to Theorem 1.13.

Corollary 1.14. *If the nonnegative square matrix A is primitive, then the matrix $\rho(A)^{-1}A$ is semi-convergent.*

1.3 DYNAMICAL SYSTEMS AND STABILITY THEORY

In this section, we introduce some basic concepts about dynamical and control systems; see, for example Sontag (1998) and Khalil (2002). We discuss stability and attractivity notions as well as the invariance principle. We conclude with a treatment of set-valued systems and time-dependent systems.

1.3.1 State machines and dynamical systems

Here, we introduce three classes of dynamical and control systems: (i) state machines or discrete-time discrete-space dynamical systems; (ii) discrete-time continuous-space control systems; and (iii) continuous-time continuous-space control systems.

We begin with our specific definition of state machine. A *(deterministic, finite) state machine* is a tuple (X, U, X_0, f) , where X is a finite set called the *state space*, U is a finite set called the *input space*, $X_0 \subset X$ is the set

of *allowable initial states*, and $f: X \times U \rightarrow X$ is the *evolution map*. Given

an input sequence $u: \mathbb{Z}_{\geq 0} \rightarrow U$, the state machine evolution $x: \mathbb{Z}_{\geq 0} \rightarrow X$

starting from $x(0) \in X_0$ is given by

$$x(\ell + 1) = f(x(\ell), u(\ell)), \quad \ell \in \mathbb{Z}_{\geq 0}.$$

We will often refer to a state machine as a *processor*. Note that, in a state

machine, both the state and the input spaces are finite or *discrete*. Often times, we will find it useful to consider systems that evolve in continuous space and that are time dependent. Let us then provide two additional definitions in the following paragraphs.

A *(time-dependent) discrete-time continuous-space control system* is a tuple (X, U, X_0, f) , where X is a d -dimensional space chosen among $\mathbf{R}^d, \mathbf{S}^d$,

and the Cartesian products $\mathbf{R}^{d_1} \times \mathbf{S}^{d_2}$, for some $d_1 + d_2 = d$, U is a com-

compact subset of $\mathbf{R}^m$ containing 0_m , $X_0 \subset X$, and $f : \mathbf{Z}_{\geq 0} \times X \times U \rightarrow X$ is a continuous map. As before, the individual objects X, U, X_0 , and f are

termed the *state space*, *input space*, *allowable initial states*, and *evolution map*, respectively. Given an input sequence $u : \mathbf{Z}_{\geq 0} \rightarrow U$, the evolution $x : \mathbf{Z}_{\geq 0} \rightarrow X$ of the dynamical system starting from $x(0) \in X_0$ is given by

$$x(\ell + 1) = f(\ell, x(\ell), u(\ell)), \quad \ell \in \mathbf{Z}_{\geq 0}.$$

A *(time-dependent) continuous-time continuous-space control system* is a tuple (X, U, X_0, f) , where X is a d -dimensional space chosen among $\mathbf{R}^d$,

$\mathbf{S}^d$, and the Cartesian products $\mathbf{R}^{d_1} \times \mathbf{S}^{d_2}$, for some $d_1 + d_2 = d$, U is a

compact subset of $\mathbf{R}^m$ containing 0_m , $X_0 \subset X$, and $f : \mathbf{R}_{\geq 0} \times X \times U \rightarrow TX$ is a continuously differentiable map. The individual objects X, U, X_0 ,

and f are termed the *state space*, *input space*, *allowable initial states*, and *control vector field*, respectively. Given an input function $u : \mathbf{R}_{\geq 0} \rightarrow U$, the evolution $x : \mathbf{R}_{\geq 0} \rightarrow X$ of the dynamical system starting from $x(0) \in X_0$ is given by

$$\dot{x}(t) = f(t, x(t), u(t)), \quad t \in \mathbf{R}_{\geq 0}.$$

We often consider the case when the control vector field can be written as

$f(t, x, u) = f_0(t, x) + \sum_{a=1}^m f_a(t, x) u_a$, for some continuously differentiable maps $f_0, f_1, \dots, f_m : \mathbf{R}_{\geq 0} \times X \rightarrow TX$. Each of these individual maps is called a *(time-dependent) vector field*, and f is said to be a *control-affine*

vector field. The control vector field f is *driftless* if $f(t, x, 0_m) = 0$ for all

$x \in X$ and $t \in \mathbf{R}_{\geq 0}$.

Finally, the term *dynamical system* denotes a control system that is not subject to any external control action; this terminology is applicable both in discrete and continuous time. Furthermore, we will sometimes neglect to define a specific set of allowable initial states; in this case we mean that any point in the state space is allowable as initial condition.

1.3.2 Stability and attractivity notions

In this section, we consider a continuous-space dynamical system (X, f) . We first consider the discrete-time case and later we briefly present the analogous continuous-time case. We study dynamical systems that are *time-invariant*. In discrete time, a time-invariant system is simply described by an evolution map of the form $f : X \rightarrow X$.

Definition 1.15 (Equilibrium point). A point $x_* \in X$ is an *equilibrium point* for the time-invariant dynamical system (X, f) if the constant curve $x : \mathbb{Z}_{\geq 0} \rightarrow X$, defined by $x(\ell) = x_*$ for all $\ell \in \mathbb{Z}_{\geq 0}$, is an evolution of the system.

.

It can immediately be seen that a point x_* is an equilibrium point if and only if $f(x_*) = x_*$. We denote the set of equilibrium points of the dynamical system by $\text{Equil}(X, f)$.

Definition 1.16 (Trajectories and sets). Let (X, f) be a time-invariant dynamical system and let W be a subset of X . Then:

- (i) The set W is *positively invariant* for (X, f) if each evolution with initial condition in W remains in W for all subsequent times.
- (ii) A trajectory $x : \mathbb{Z}_{\geq 0} \rightarrow X$ *approaches* a set $W \subset X$ if, for every neighborhood Y of W , there exists a time $\ell_0 > 0$ such that $x(\ell)$ takes values in Y for all subsequent times $\ell \geq \ell_0$. In such a case, we write $x(\ell) \rightarrow W$ as $\ell \rightarrow +\infty$.

In formal terms, W is positively invariant if $x(0) \in W$ implies $x(\ell) \in W$ for all $\ell \in \mathbb{Z}_{\geq 0}$, where $x : \mathbb{Z}_{\geq 0} \rightarrow X$ is the evolution of (X, f) starting from $x(0)$.

Definition 1.17 (Stability and attractivity). For a time-invariant dynamical system (X, f) , a set S is:

- (i) *stable* if, for any neighborhood Y of S , there exists a neighborhood W of S such that every evolution of (X, f) with initial condition in

W remains in Y for all subsequent times;

- (ii) *unstable* if it is not stable;
- (iii) *locally attractive* if there exists a neighborhood Y of S such that every evolution with initial condition in Y approaches the set S ; and
- (iv) *locally asymptotically stable* if it is stable and locally attractive.

Additionally, the set S is *globally attractive* if every evolution of the dynamical system approaches it and it is *globally asymptotically stable* if it is stable and globally attractive.

Remark 1.18 (Continuous-time dynamical systems). It is straightforward to extend the previous definitions to the setting of continuous-time continuous-space dynamical systems. These notions are illustrated in Figure 1.2.

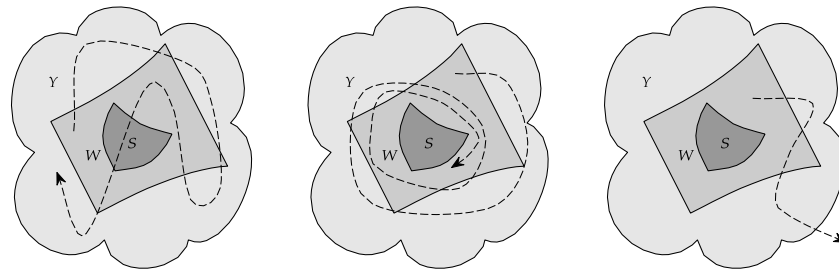


Figure 1.2 Illustrations of stability, asymptotic stability, and instability.

1.3.3 Invariance principles

Before discussing various versions of the invariance principle, we begin with a useful notion. Given a discrete-time time-invariant continuous-space dynamical system (X, f) and a set $W \subseteq X$, a function $V : X \rightarrow \mathbb{R}$ is *non-*

increasing along f in W if $V(f(x)) \leq V(x)$ for all $x \in W$. (Such functions

are often referred to as *Lyapunov functions*.) In other words, if a function V is non-increasing along f , then the composite function $\ell \mapsto V(y(\ell))$ is non-increasing for each evolution y of the dynamical system (X, f) . The following theorem exploits this fact to establish useful properties of the evolutions of (X, f) .

Theorem 1.19 (LaSalle Invariance Principle for discrete-time dynamical systems). *Let (X, f) be a discrete-time time-independent dynamical system. Assume that:*

- (i) there exists a closed set $W \subset X$ that is positively invariant for (X, f) ;*
- (ii) there exists a function $V : X \rightarrow \mathbb{R}$ that is non-increasing along f on W ;*
- (iii) all evolutions of (X, f) with initial conditions in W are bounded; and*
- (iv) f and V are continuous on W .*

Then each evolution with initial condition in W approaches a set of the form $V^{-1}(c) \cap S$, where c is a real constant and S is the largest positively invariant set contained in $\{w \in W \mid V(f(w)) = V(w)\}$.

We refer to Section 1.8.1 for a discussion about the proof of this result. Next, we present the continuous-time version of the invariance principle. In other words, we now assume that (X, f) is a continuous-time time-invariant continuous-space dynamical system.

We begin by revisiting the notion of non-increasing function. Given a continuously differentiable function $V : X \rightarrow \mathbb{R}$, the *Lie derivative* of V along f , denoted by $L_f V : X \rightarrow \mathbb{R}$, is defined by

$$L_f V(x) = \frac{d}{dt} V(\gamma(t)) \Big|_{t=0},$$

where the trajectory $\gamma :]-\varepsilon, \varepsilon[\rightarrow X$ satisfies $\dot{\gamma}(t) = f(\gamma(t))$ and $\gamma(0) = x$. If $X = \mathbb{R}^d$, then we can write x in components $(x_1, \dots, x_d)$ and we can give the following explicit formula for the Lie derivative:

$$L_f V(x) = \sum_{i=1}^d \frac{\partial V}{\partial x_i}(x) f_i(x).$$

Similar formulas can be obtained for more general state spaces. Note that, given a set $W \subset X$, a function $V : X \rightarrow \mathbb{R}$ is non-increasing along f in W if $L_f V(x) \leq 0$ for all $x \in W$.

Finally, we state the invariance principle for continuous-time systems.

Theorem 1.20 (LaSalle Invariance Principle for continuous-time dynamical systems). *Let (X, f) be a continuous-time time-independent dynamical system. Assume that:*

- (i) *there exists a closed set $W \subset X$ that is positively invariant for (X, f) ;*
- (ii) *there exists a function $V : X \rightarrow \mathbb{R}$ that is non-increasing along f*

on W ;

(iii) all evolutions of (X, f) with initial conditions in W are bounded;
and

(iv) f and V are continuously differentiable⁴ on W .

Then, each evolution with initial condition in W approaches a set of the form $V^{-1}(c) \cap S$, where c is a real constant and S is the largest positively invariant set contained in $\{w \in W \mid L_f V(w) = 0\}$.

⁴It suffices that f be locally Lipschitz and V be continuously differentiable; see [Cortés \(2008a\)](#).

1.3.4 Notions and results for set-valued systems

Next, we focus on a more sophisticated version of the LaSalle Invariance Principle for more general dynamical systems, that is, dynamical systems described by set-valued maps that allow for non-deterministic evolutions. To do so, we need to present numerous notions, including set-valued dynamical systems, closedness properties, and weak positive invariance.

Specifically, a *discrete-time continuous-space set-valued dynamical system* (in short, *set-valued dynamical system*) is determined by a tuple (X, X_0, T) , where X is a d -dimensional space chosen among $\mathbb{R}^d$, S^d , and the Cartesian

products $\mathbb{R}^{d_1} \times S^{d_2}$, for some $d_1 + d_2 = d$, $X_0 \subset X$, and $T : X \Rightarrow X$ is a set-valued map. We assume that T assigns to each point $x \in X$ a nonempty set $T(x) \subset X$. The individual objects X , X_0 , and T are termed the *state space*, *allowable initial states*, and *evolution map*, respectively. An *evolution*

of the dynamical system (X, X_0, T) is any trajectory $x : \mathbb{Z}_{\geq 0} \rightarrow X$ satisfying

$$x(\ell + 1) \in T(x(\ell)), \quad \ell \in \mathbb{Z}_{\geq 0}.$$

Figure 1.3 illustrates this notion. In particular, a (time-invariant) discrete-time continuous-space dynamical system (X, X_0, f) can be seen as a discrete-time continuous-space set-valued dynamical system (X, X_0, T) , where the evolution set-valued map is just the singleton-valued map $x \mapsto T(x) = \{f(x)\}$.

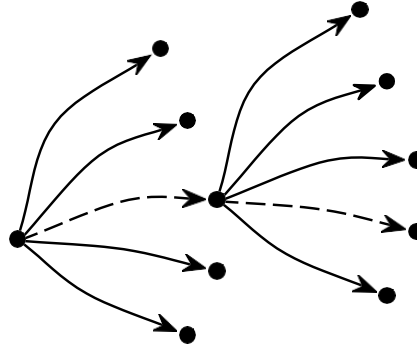


Figure 1.3 A discrete-time continuous-space set-valued dynamical system. A sample evolution is shown dashed.

Next, we introduce a notion of continuity for set-valued maps. The evolution map T is said to be *closed at* $x \in X$ if, for any sequences $\{x_k \mid k \in \mathbb{Z}_{\geq 0}\}$

and $\{y_k \mid k \in \mathbb{Z}_{\geq 0}\}$ such that

$$\lim_{k \rightarrow +\infty} x_k = x, \quad \lim_{k \rightarrow +\infty} y_k = y, \quad \text{and} \quad y_k \in T(x_k),$$

it holds that $y \in T(x)$. The evolution set-valued map T is *closed at* $W \subset X$ if for any $x \in W$, T is closed at x . Note that a continuous map $f : X \rightarrow X$ is closed when viewed as a singleton-valued map.

- (i) A set $C \subset X$ is *weakly positively invariant with respect to* T if, for any $x \in C$, there exists $y \in C$ such that $y \in T(x)$.
- (ii) A set $C \subset X$ is *strongly positively invariant with respect to* T if $T(x) \subset C$ for any $x \in C$.

A point x_0 is said to be a *fixed point of* T if $x_0 \in T(x_0)$. A continuous function $V : X \rightarrow \mathbb{R}$ is *non-increasing along* T in $W \subset X$ if $V(y) \leq V(x)$ for all $x \in W$ and $y \in T(x)$.

We finally state and prove a general version of the invariance principle, whose proof is presented in Section 1.8.1.

Theorem 1.21 (LaSalle Invariance Principle for set-valued discrete-time dynamical systems). *Let (X, X_0, T) be a discrete-time set-valued dynamical system. Assume that:*

- (i) *there exists a closed set $W \subset X$ that is strongly positively invariant for (X, X_0, T) ;*
- (ii) *there exists a function $V : X \rightarrow \mathbb{R}$ that is non-increasing along T on W ;*
- (iii) *all evolutions of (X, X_0, T) with initial conditions in W are bounded; and*
- (iv) *T is nonempty and closed at W and V is continuous on W .*

Then, each evolutions with initial condition in W approaches a set of the form $V^{-1}(c) \cap S$, where c is a real constant and S is the largest weakly positively invariant set contained in $\{w \in W \mid \exists w' \in T(w) \text{ such that } V(w') = V(w)\}$.

1.3.5 Notions and results for time-dependent systems

In this final subsection, we consider time-dependent discrete-time dynamical systems and discuss *uniform* stability and convergence notions. We begin with some uniform boundedness, stability, and attractivity definitions.

In what follows, given a time-dependent discrete-time dynamical system (X, X_0, f) , an evolution with initial condition in W at time $\ell_0 \in \mathbb{Z}_{\geq 0}$ is a trajectory $x : [\ell_0, +\infty[\rightarrow X$ of the dynamical system (X, X_0, f) defined by the initial condition $x(\ell_0) = x_0$, for some $x_0 \in W$. In other words, for time-dependent systems we will often consider trajectories that begin at time ℓ_0 not necessarily equal to zero.

Definition 1.22 (Uniformly bounded evolutions). A time-dependent discrete-time dynamical system (X, X_0, f) has *uniformly bounded evolutions* if, given any bounded set Y , there exists a bounded set W such that every evolution with initial condition in Y at any time $\ell_0 \in \mathbb{Z}_{\geq 0}$ remains in W for all subsequent times $\ell \geq \ell_0$. •

Definition 1.23 (Uniform stability and attractivity notions). For a time-dependent discrete-time dynamical system (X, X_0, f) , the set S is:

- (i) *uniformly stable* if, for any neighborhood Y of S , there exists a neighborhood W of S such that every evolution with initial condition in W at any time $\ell_0 \in \mathbb{Z}_{\geq 0}$ remains in Y for all subsequent times $\ell \geq \ell_0$;
- (ii) *uniformly locally attractive* if there exists a neighborhood Y of S such that every evolution with initial condition in Y at any time ℓ_0 approaches the set S in the following *time-uniform* manner:
for all $\ell_0 \in \mathbb{Z}_{\geq 0}$, for all $x_0 \in Y$, and for all neighborhoods W of S , there exists a single $\tau_0 \in \mathbb{Z}_{\geq 0}$ such that the evolution $x : [\ell_0, +\infty[\rightarrow X$ defined by $x(\ell_0) = x_0$ takes value in W for all times $\ell \geq \ell_0 + \tau_0$; and
- (iii) *uniformly locally asymptotically stable* if it is uniformly stable and uniformly locally attractive.

Additionally, the set S is *uniformly globally attractive* if every evolution of the dynamical system approaches the set in a time-uniform manner, and it is *uniformly globally asymptotically stable* if it is uniformly stable and uniformly globally attractive. •

With the same notation in the definition, the set S is *(non-uniformly) locally attractive* if for all $\ell_0 \in \mathbb{Z}_{\geq 0}$, $x_0 \in Y$, and neighborhoods W of S , the evolution $x : [\ell_0, +\infty[\rightarrow X$ defined by $x(\ell_0) = x_0$, takes value in W for all times $\ell \geq \ell_0 + \tau_0(\ell_0)$, for some $\tau_0(\ell_0) \in \mathbb{Z}_{\geq 0}$.

To establish uniform stability and attractivity results we will overapproximate the evolution of the time-dependent dynamical system by considering the larger set of evolutions of an appropriate set-valued dynamical system. Given a time-dependent evolution map $f : \mathbb{Z}_{\geq 0} \times X \rightarrow X$, define a set-valued

overapproximation map $T_f : X \rightrightarrows X$ by

$$T_f(x) = \{f(\ell, x) \mid \ell \in \mathbb{Z}_{\geq 0}\}.$$

With this notion we can state a useful result, whose proof is left to the reader as an exercise.

Lemma 1.24 (Overapproximation Lemma). *Consider a discrete-time time-dependent dynamical system (X, X_0, f) :*

- (i) *If $x : [\ell_0, +\infty[\rightarrow X$ is an evolution of the dynamical system (X, f) , then $y : \mathbb{Z}_{\geq 0} \rightarrow X$ defined by $y(\ell) = x(\ell + \ell_0)$ is an evolution of the set-valued overapproximation system (X, T_f) .*
- (ii) *If the set S is locally attractive for the set-valued overapproximation system (X, T_f) , then it is uniformly locally attractive for (X, f) .*

In other words, every evolution of the time-dependent dynamical system from any initial time is an evolution of the set-valued overapproximation system and, therefore, the set of trajectories of the set-valued overapproximation system contains the set of trajectories of the original time-dependent system. Uniform attractivity is a consequence of attractivity for the time-invariant set-valued overapproximation.

1.4 GRAPH THEORY

Here we present basic definitions about graph theory, following the treatments in the literature; see, for example Biggs (1994), Godsil and Royle (2001), and Diestel (2005).

A *directed graph*—in short, *digraph*—of order n is a pair $tt = (V, E)$, where V is a set with n elements called *vertices* (or *nodes*) and E is a set of ordered pair of vertices called *edges*. In other words, $E \subseteq V \times V$. We call V and E the *vertex set* and *edge set*, respectively. When convenient, we let $V(tt)$ and $E(tt)$ denote the vertices and edges of tt , respectively. For $u, v \in V$, the ordered pair (u, v) denotes an edge from u to v .

An *undirected graph*—in short, *graph*—consists of a vertex set V and of a set E of unordered pairs of vertices. For $u, v \in V$ and $u \neq v$, the set $\{u, v\}$

denotes an unordered edge. A digraph is *undirected* if $(v, u) \in E$ anytime $(u, v) \in E$. It is possible and convenient to identify an undirected digraph with the corresponding graph; vice versa, the *directed version* of a graph

(V, E) is the digraph (V', E') with the property that $(u, v) \in E'$ if and only if $(u, v) \in E$ or $(v, u) \in E$. In what follows, our convention is to allow self-loops in both graphs and digraphs.

A digraph (V', E') is a *subgraph* of a digraph (V, E) if $V' \subseteq V$ and $E' \subseteq E$; additionally, a digraph (V', E') is a *spanning subgraph* if it is a subgraph and $V' = V$. The subgraph of (V, E) *induced by* $V' \subseteq V$ is the digraph (V', E') , where E' contains all edges in E between two vertices in V' . For two digraphs $tt = (V, E)$ and $tt' = (V', E')$, the *intersection* and *union* of tt and tt' are defined by

$$\begin{aligned} tt \cap tt' &= (V \cap V', E \cap E'), \\ tt \cup tt' &= (V \cup V', E \cup E'). \end{aligned}$$

Analogous definitions may be given for graphs.

In a digraph tt with an edge $(u, v) \in E$, u is called an *in-neighbor* of v , and v is called an *out-neighbor* of u . We let $N_{tt}^{\text{in}}(v)$ (resp., $N_{tt}^{\text{out}}(v)$) denote the set of in-neighbors, (resp. the set of out-neighbors) of v in the digraph tt . We will drop the subscript when the graph tt is clear from the context. The *in-degree* and *out-degree* of v are the cardinality of $N_{tt}^{\text{in}}(v)$ and $N_{tt}^{\text{out}}(v)$, respectively. A digraph is *topologically balanced* if each vertex has the same in- and out-degrees (even if distinct vertices have distinct degrees). Likewise, in an undirected graph tt , the vertices u and v are *neighbors* if $\{u, v\}$ is an undirected edge. We let $N_{tt}(v)$ denote the set of neighbors of v in the undirected graph tt . As in the directed case, we will drop the subscript when the graph tt is clear from the context. The *degree* of v is the cardinality of $N_{tt}(v)$.

Remark 1.25 (Additional notions). For a digraph $tt = (V, E)$, the *re-verse digraph* $\text{rev}(tt)$ has vertex set V and edge set $\text{rev}(E)$ composed of all edges in E with reversed direction. A digraph $tt = (V, E)$ is *complete* if

$E = V \times V$. A *clique* (V', E') of a digraph (V, E) is a subgraph of (V, E) which is complete, that is, such that $E' = V' \times V'$. Note that a clique is fully determined by its set of vertices, and hence there is no loss of precision in denoting it by V' . A *maximal clique* V' of an edge of a digraph is a clique of the digraph with the following two properties: it contains the edge, and any other subgraph

of the digraph that strictly contains $(V', V' \times V')$ is not a clique.

•

1.4.1 Connectivity notions

Let us now review some basic connectivity notions for digraphs and graphs. We begin with the setting of undirected graphs because of its simplicity.

A *path* in a graph is an ordered sequence of vertices such that any pair of consecutive vertices in the sequence is an edge of the graph. A graph is *connected* if there exists a path between any two vertices. If a graph is not

connected, then it is composed of multiple *connected components*, that is, multiple connected subgraphs. A path is *simple* if no vertices appear more than once in it, except possibly for initial and final vertex. A *cycle* is a simple path that starts and ends at the same vertex. A graph is *acyclic* if it contains no cycles. A connected acyclic graph is a *tree*. A *forest* is a graph that can be written as the disjoint union of trees. Trees have interesting

properties: for example, $tt = (V, E)$ is a tree if and only if tt is connected and $|E| = |V| - 1$. Alternatively, $tt = (V, E)$ is a tree if and only if tt is acyclic and $|E| = |V| - 1$. Figure 1.4 illustrates these notions.

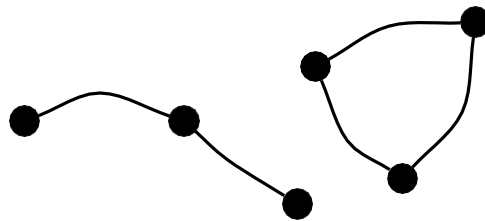


Figure 1.4 An illustration of connectivity notions on a graph. The graph has two connected components. The leftmost connected component is a tree, while the rightmost connected component is a cycle.

Next, we generalize these notions to the case of digraphs. A *directed path* in a digraph is an ordered sequence of vertices such that any ordered pair of vertices appearing consecutively in the sequence is an edge of the digraph. A *cycle* in a digraph is a directed path that starts and ends at the same vertex and that contains no repeated vertex except for the initial and the final vertex. A digraph is *acyclic* if it contains no cycles. In an acyclic graph, every vertex of in-degree 0 is named a *source*, and every vertex of out-degree 0 is named a *sink*. Every acyclic digraph has at least one source and at least one sink. Figure 1.5 illustrates these notions.

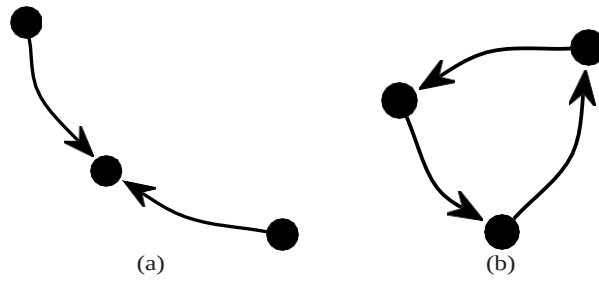


Figure 1.5 Illustrations of connectivity notions on a digraph: (a) shows an acyclic digraph with one sink and two sources; (b) shows a directed path which is also a cycle.

The set of cycles of a directed graph is finite. A directed graph is *aperiodic* if there exists no $k > 1$ that divides the length of every cycle of the graph.

In other words, a digraph is aperiodic if the greatest common divisor of the lengths of its cycles is one. A digraph is *periodic* if it is not aperiodic. Figure 1.6 shows examples of a periodic and an aperiodic digraph.

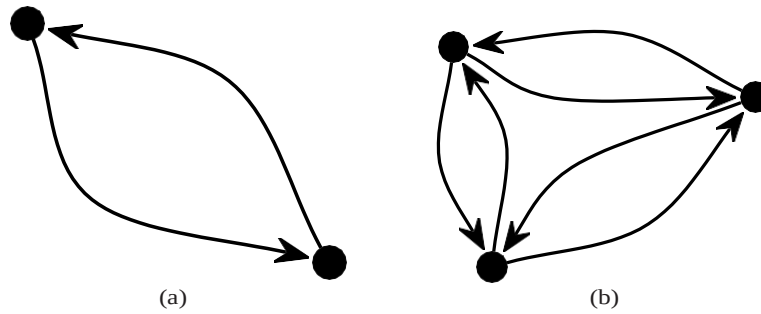


Figure 1.6 (a) A periodic digraph. (b) An aperiodic digraph with cycles of length 2 and 3.

A vertex of a digraph is *globally reachable* if it can be reached from any other vertex by traversing a directed path. A digraph is *strongly connected* if every vertex is globally reachable. The decomposition of a digraph into its strongly connected components and the notion of condensation digraph are discussed in Exercise E1.13.

A *directed tree* (sometimes called a rooted tree) is an acyclic digraph with the following property: there exists a vertex, called the *root*, such that any other vertex of the digraph can be reached by one and only one directed path starting at the root. In a directed tree, every in-neighbor of a vertex is called a *parent* and every out-neighbor is called a *child*. Two vertices with the same parent are called *siblings*. A *successor* of a vertex u is any other node that can be reached with a directed path starting at u . A *predecessor* of a vertex v is any other node such that a directed path exists starting at it and reaching v . A *directed spanning tree*, or simply a *spanning tree*, of a digraph is a spanning subgraph that is a directed tree.

Clearly, a digraph contains a spanning tree if and only if the reverse digraph contains a globally reachable vertex. A *(directed) chain* is a directed tree with exactly one source and one sink. A *(directed) ring digraph* is the cycle obtained by adding to the edge set of a chain a new edge from its sink to its source. Figure 1.7 illustrates some of these notions.

The proof of the following result is given in Section 1.8.2.

Lemma 1.26 (Connectivity in topologically balanced digraphs). *Let G be a digraph. The following statements hold:*

- (i) *if G is strongly connected, then it contains a globally reachable vertex*

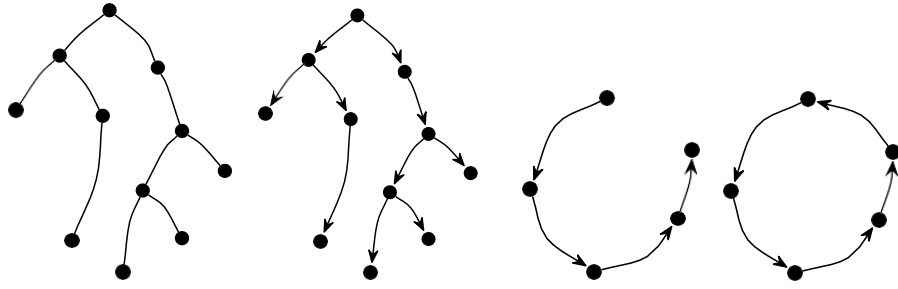


Figure 1.7 From left to right, tree, directed tree, chain, and ring digraphs.

and a spanning tree; and

- (ii) if tt is topologically balanced and contains either a globally reachable vertex or a spanning tree, then tt is strongly connected and is Eulerian.⁵

Given a digraph $tt = (V, E)$, an *in-neighbor* of a nonempty set of nodes U is a node $v \in V \setminus U$ for which there exists an edge $(v, u) \in E$ for some $u \in U$.

Lemma 1.27 (Disjoint subsets and spanning trees). *Given a digraph tt with at least two nodes, the following two properties are equivalent:*

- (i) tt has a spanning tree; and
- (ii) for any pair of nonempty disjoint subsets $U_1, U_2 \subset V$, either U_1 has an in-neighbor or U_2 has an in-neighbor.

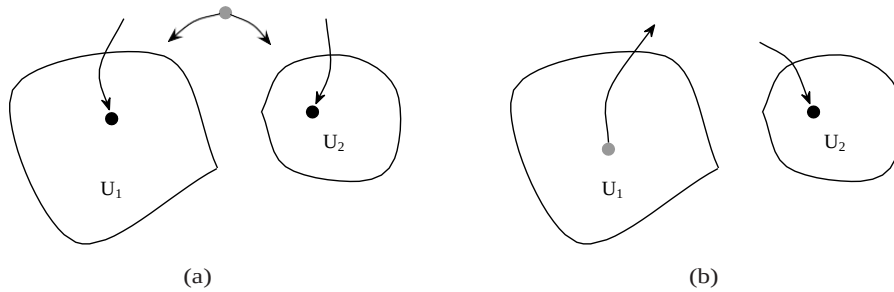


Figure 1.8 An illustration of Lemma 1.27. The root of the spanning tree is plotted in gray. In (a), the root is outside the sets U_1 and U_2 . Because these sets are non-empty, there exists a directed path from the root to a vertex in each one of these sets. Therefore, both U_1 and U_2 have in-neighbors. In (b), the root is contained in U_1 . Because U_2 is non-empty, there exists a directed path from the root to a vertex in U_2 , and, therefore, U_2 has in-neighbors. The case when the root belongs to U_2 is treated analogously.

⁵A graph is Eulerian if it has a cycle that visits all the graph edges exactly once.

We will postpone the proof to Section 1.8.2. The result is illustrated in Figure 1.8. We can also state the result in terms of global reachability: tt has a globally reachable node if and only if, for any pair of nonempty disjoint subsets $U_1, U_2 \subseteq V$, either U_1 has an out-neighbor or U_2 has an out-neighbor. We let the reader give a proper definition of the out-neighbor of a set.

1.4.2 Weighted digraphs

A *weighted digraph* is a triplet $tt = (V, E, A)$, where the pair (V, E) is a digraph with nodes $V = \{v_1, \dots, v_n\}$, and where the nonnegative matrix $A \in \mathbb{R}_{n \times n}$ is a *weighted adjacency matrix* with the following property: for

$i, j \in \{1, \dots, n\}$, the entry $a_{ij} \geq 0$ if (v_i, v_j) is an edge of tt , and $a_{ij} = 0$ otherwise. In other words, the scalars a_{ij} , for all $(v_i, v_j) \in E$, are a set of weights for the edges of tt . Note that the edge set is uniquely determined by the weighted adjacency matrix and it can therefore be omitted. When convenient, we denote the adjacency matrix of a weighted digraph tt by $A(tt)$. Figure 1.9 shows an example of a weighted digraph.

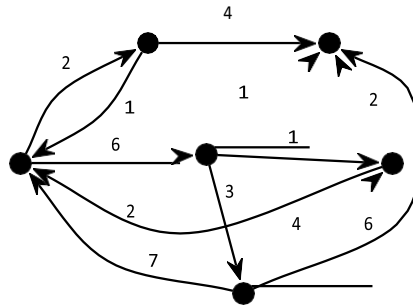


Figure 1.9 A weighted digraph with natural weights.

A digraph $tt = (V, E)$ can be naturally thought of as a weighted digraph by defining the weighted adjacency matrix $A \in \{0, 1\}^{n \times n}$ as

$$a_{ij} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

0, otherwise,

$$(1.4.1)$$

where $V = \{v_1, \dots, v_n\}$. The adjacency matrix of a graph is the adjacency matrix of the directed version of the graph. Reciprocally, given a weighted digraph $tt = (V, E, A)$, we refer to the digraph (V, E) as the *unweighted version* of tt and to its associated adjacency matrix as the *unweighted adjacency matrix*. A weighted digraph is *undirected* if $a_{ij} = a_{ji}$ for all $i, j \in \{1, \dots, n\}$. Clearly, tt is undirected if and only if $A(tt)$ is symmetric.

Numerous concepts introduced for digraphs remain equally valid for the case of weighted digraphs, including the connectivity notions and the definitions of in- and out-neighbors.

Finally, we generalize the notions of in- and out-degree to weighted digraphs. In a weighted digraph $tt = (V, E, A)$ with $V = \{v_1, \dots, v_n\}$, the *weighted out-degree* and the *weighted in-degree* of vertex v_i are defined by, respectively,

$$d_{\text{out}}(v_i) = \sum_{j=1}^n a_{ij}, \quad \text{and} \quad d_{\text{in}}(v_i) = \sum_{j=1}^n a_{ji}.$$

The weighted digraph tt is *weight-balanced* if $d_{\text{out}}(v_i) = d_{\text{in}}(v_i)$ for all $v_i \in V$. The *weighted out-degree matrix* $D_{\text{out}}(tt)$ and the *weighted in-degree matrix*

$D_{\text{in}}(tt)$ are the diagonal matrices defined by

$$D_{\text{out}}(tt) = \text{diag}(A1_n), \quad \text{and} \quad D_{\text{in}}(tt) = \text{diag}(A^T 1_n).$$

That is, $(D_{\text{out}}(tt))_{ii} = d_{\text{out}}(v_i)$ and $(D_{\text{in}}(tt))_{ii} = d_{\text{in}}(v_i)$, respectively.

1.4.3 Distances on digraphs and weighted digraphs

We first present a few definitions for unweighted digraphs. Given a digraph tt , the *(topological) length* of a directed path is the number of the edges composing it. Given two vertices u and v in the digraph tt , the *distance*

from u to v , denoted $\text{dist}_{tt}(u, v)$, is the smallest length of any directed path from u to v , or $+\infty$ if there is no directed path from u to v . That is,

$\text{dist}_{tt}(u, v) = \min \{\text{length}(p) \mid p \text{ is a directed path from } u \text{ to } v\} \cup \{+\infty\}$. Given a vertex v of a digraph tt , the *radius* of v in tt is the maximum of all the distances from v to any other vertex in tt . That is,

$$\text{radius}(v, tt) = \max\{\text{dist}_{tt}(v, u) \mid u \in V(tt)\}.$$

If T is a directed tree and v is its root, then the *depth* of T is $\text{radius}(v, T)$.

Finally, the *diameter* of the digraph tt is

$$\text{diam}(tt) = \max\{\text{dist}_{tt}(u, v) \mid u, v \in V(tt)\}.$$

These definitions lead to the following simple results:

- (i) $\text{radius}(v, tt) \leq \text{diam}(tt)$ for all vertices v of tt ;
- (ii) tt contains a spanning tree rooted at v if and only if $\text{radius}(v, tt) < +\infty$; and
- (iii) tt is strongly connected if and only if $\text{diam}(tt) < +\infty$.

The definitions of path length, distance between vertices, radius of a vertex, and diameter of a digraph can be easily applied to undirected graphs.

Next, we consider weighted digraphs. Given two vertices u and v in the weighted digraph tt , the *weighted distance* from u to v , denoted $\text{wdist}_{tt}(u, v)$, is the smallest weight of any directed path from u to v , or $+\infty$ if there is no directed path from u to v . That is,

$\text{wdist}_{tt}(u, v) = \min \{ \text{weight}(p) \mid p \text{ is a directed path from } u \text{ to } v \} \cup \{+\infty\}$. Here, the weight of a subgraph of a weighted digraph is the sum of the weights of all the edges of the subgraph. Note that when a digraph is thought of as a weighted digraph (with the unweighted adjacency matrix (1.4.1)), the notions of weight and weighted distance correspond to the usual notions of length and distance, respectively. We leave it the reader to provide the definitions of weighted radius, weighted depth, and weighted diameter.

1.4.4 Graph algorithms

In this section, we present a few algorithms defined on graphs. We present only high-level descriptions and we refer to [Cormen et al. \(2001\)](#) for a comprehensive discussion including a detailed treatment of computationally efficient data structures and algorithmic implementations.

1.4.4.1 Breadth-first spanning tree

Let v be a vertex of a digraph tt with $\text{radius}(v, tt) < +\infty$. A *breadth-first spanning (BFS) tree* of tt with respect to v , denoted T_{BFS} , is a spanning directed tree rooted at v that contains a shortest path from v to every other

vertex of tt . (Here, a shortest path is one with the shortest topological length.) Let us provide the BFS algorithm that, given a digraph tt of order n and a vertex v with $\text{radius}(v, tt) < +\infty$, computes a BFS tree T_{BFS} rooted at v :

[Informal description] Initialize a subgraph to contain only the root v . Repeat $\text{radius}(v, tt)$ times the following instructions: attach to the subgraph all out-neighbors of the subgraph as well as a single connecting edge for each out-neighbor. The final subgraph is the desired directed tree.

The algorithm is formally stated as follows:

function BFS(tt, v)

- 1: $(V_1, E_1) := (\underbrace{v}_{\wedge}, \{ \})$
- 2: for $k = 2$ to $\text{radius}(v, tt)$ do
- 3: find all vertices $w_1, \dots, w_m$ not in V_{k-1} that are out-neighbors of some vertex in V_{k-1} and, for $j \in \{1, \dots, m\}$, let e_j be an edge connecting a vertex in V_{k-1} to w_j
- 4: $V_k := V_{k-1} \cup \{w_1, \dots, w_m\}$
- 5: $E_k := E_{k-1} \cup \{e_1, \dots, e_m\}$
- 6: return (V_n, E_n)

Note that the output of this algorithm is not necessarily unique, since the choice of edges at step 3: in the algorithm is not unique. Figure 1.10 shows an execution of the BFS algorithm.

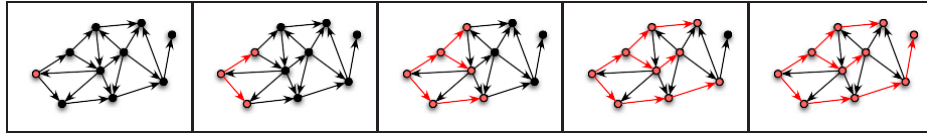


Figure 1.10 Execution of the BFS algorithm. In the leftmost frame, vertex v is colored in red. The other frames correspond to incremental additions of vertices and edges as specified by the function BFS. The output of the algorithm is a BFS tree of the digraph. The BFS tree is represented in the last frame with vertices and edges colored in red.

Some properties of the BFS algorithm are characterized as follows.

Lemma 1.28 (BFS tree). *For a digraph tt with a vertex v , any digraph T computed by the BFS algorithm, $T \in \text{BFS}(tt, v)$, has the following properties:*

- (i) T is a directed tree with root v ;
- (ii) T contains a shortest path from v to any other vertex reachable from v inside tt , that is, if there is a path in tt from v to w , then $w \in T$ and $\text{dist}_{tt}(v, w) = \text{dist}_T(v, w)$; and
- (iii) if tt contains a spanning tree rooted at v , then T is spanning too and therefore, T is a BFS tree of tt .

We leave the proof to the reader. The key property of the

algorithm is that

(V_k, E_k) , $k \in \{1, \dots, n\}$, is a sequence of directed trees with the property that $(V_k, E_k) \subset (V_{k+1}, E_{k+1})$, for $k \in \{1, \dots, n-1\}$.

1.4.4.2 The depth-first spanning tree

Next, we define the DFS algorithm that, given a digraph tt and a vertex v with $\text{radius}(v, tt) < +\infty$, computes what we term a *depth-first spanning (DFS) tree* T_{DFS} rooted at v :

[Informal description] Visit all nodes of the graph recording the traveled edges to form the desired tree. Visit the nodes in the following recursive way: (1) as long as a node has an unvisited child, visit it; (2) when the node has no more unvisited children, then return to its parent (and recursively attempt to visit its unvisited children).

The algorithm is formally stated as a recursive procedure, as follows:

```
function DFS( $tt, v$ )
1: ( $V_{\text{visited}}, E_{\text{visited}}$ ) := ( $\{v\}, \emptyset$ )
2: DFS-Visit( $tt, v$ )
3: return ( $V_{\text{visited}}, E_{\text{visited}}$ )
```

```
function DFS-Visit( $tt, w$ )
1: for  $u$  out-neighbor of  $w$  do
2: if  $u$  does not belong to  $V_{\text{visited}}$  then
3:    $V_{\text{visited}} := V_{\text{visited}} \cup \{u\}$ 
4:    $E_{\text{visited}} := E_{\text{visited}} \cup \{(w, u)\}$ 
5:   DFS-Visit( $tt, u$ )
```

Note that the output of this algorithm is not necessarily unique, since the order in which the vertices are chosen in step 1: of DFS-Visit is not unique. Any digraph T computed by the DFS algorithm, $T = \text{DFS}(tt, v)$, is a directed spanning tree with root v . Figure 1.11 shows an execution of the algorithm.

Some properties of the DFS algorithm are characterized as follows.

Lemma 1.29 (DFS tree). *For a digraph tt with a vertex v , any digraph T computed by the DFS algorithm, $T \in \text{DFS}(tt, v)$, has the following properties:*

- (i) *T is a directed tree with root v ; and*
- (ii) *if tt contains a spanning tree rooted at v , then T is spanning too.*

Note that both BFS and DFS trees are uniquely defined once a lexico- graphic order is introduced for the children of a node.

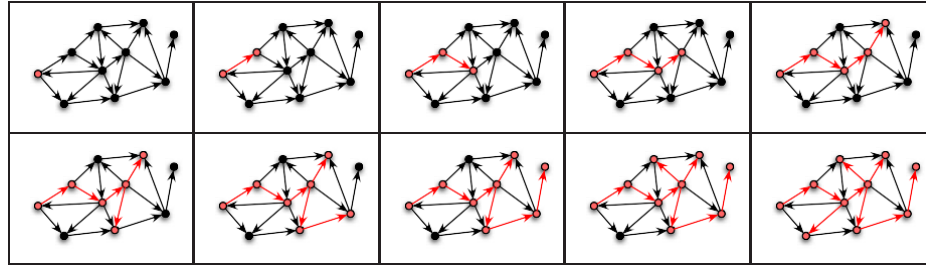


Figure 1.11 Execution of the DFS algorithm. In the top leftmost frame, vertex v is colored in red. The other frames correspond to incremental additions of vertices and edges as specified by the function DFS. The output of the algorithm is a DFS tree of the digraph. The DFS tree is represented in the last frame, with vertices and edges in red.

1.4.4.3 The shortest-paths tree in weighted digraphs via the Dijkstra algorithm

Finally, we focus on weighted digraphs and on the notion of weighted path length. Given a weighted digraph tt of order n with weighted adjacency matrix A and a vertex v with $\text{radius}(v, tt) < +\infty$, a *shortest-paths tree* of tt with respect to v , denoted $T_{\text{shortest-paths}}$, is a spanning directed tree rooted

at v that contains a (weighted) shortest path from v to every other vertex of tt . This tree differs from the BFS tree defined above because here the path length is measured using the digraph weights.

We now provide the Dijkstra algorithm that, given a digraph tt of order n and a vertex v with $\text{radius}(v, tt) < +\infty$, computes a shortest-paths tree $T_{\text{shortest-paths}}$ rooted at v :

[Informal description] Incrementally construct a tree that contains only shortest paths. In each round, add to the tree (1) the node that is closest to the source and is not yet in the tree, and (2) the edge corresponding to the shortest path. The weighted distance to the source (required to perform step (1)) is computed via an array of

distance estimates that is updated as follows:
 when a node is added to the tree, the distance
 estimates of all its out-neighbors are updated.

The algorithm is formally stated as follows:

```

function Dijkstra( $V, E, A, v$ )
1:  $T := \emptyset$ 
   % Initialize estimated distances and estimated parent nodes
2: for  $u \in V$  do
  
```



```

3:   $\text{dist}(u) := \begin{cases} 0, & u = v, \\ +\infty, & \text{otherwise.} \end{cases}$ 

4:   $\text{parent}(u) := u$ 

   % Main loop to grow the tree and update estimates

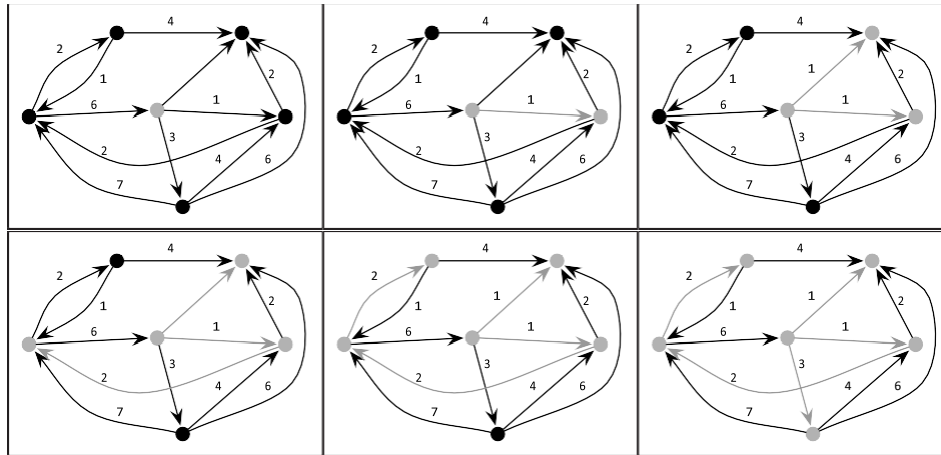
5:  while ( $T_{\text{shortest-paths}}$  does not contain all vertices) do
6:    find vertex  $u$  outside  $T_{\text{shortest-paths}}$  with smallest  $\text{dist}(u)$ 
7:    add to  $T_{\text{shortest-paths}}$  the vertex  $u$ 
8:    if  $u = v$ , add to  $T_{\text{shortest-paths}}$  the edge  $(\text{parent}(u), u)$ 
9:    for each node  $w$  that is an out-neighbor of  $u$  in  $(V, E, A)$  do

10:     if  $\text{dist}(w) > \text{dist}(u) + a_{uw}$  then
11:        $\text{dist}(w) := \text{dist}(u) + a_{uw}$ 
12:        $\text{parent}(w) := u$ 

13: return  $T_{\text{shortest-paths}}$ 

```

Note that the output of this algorithm is not necessarily unique, since the choice of vertex at step 6: in the algorithm is not unique. Figure 1.12 shows an execution



of the the Dijkstra algorithm.

Figure 1.12 Execution of the Dijkstra algorithm on the weighted digraph plotted in Figure 1.9. In the top leftmost frame, vertex v is colored in gray. The other frames correspond to incremental additions of vertices and edges as specified by the function Dijkstra. The output of the algorithm is a shortest-paths tree of the digraph rooted at v . This tree is represented in the last frame with vertices and edges colored in gray.

The following properties of the Dijkstra algorithm mirror

those of the
BFS algorithm in Lemma 1.28.

Lemma 1.30 (Dijkstra algorithm). *For a weighted digraph tt with a vertex v , any digraph T computed by the Dijkstra algorithm, $T \in \text{Dijkstra}(tt, v)$, has the following properties:*

- (i) T is a directed tree with root v ;
- (ii) T contains a shortest path from v to any other vertex reachable from v inside tt , that is, if there is a path in tt from v to w , then $w \in T$ and $\text{wdist}_{tt}(v, w) = \text{wdist}_T(v, w)$; and
- (iii) if tt contains a spanning tree rooted at v , then T is spanning too, and therefore, T is a shortest-paths tree of tt .

1.4.4.4 On combinatorial optimization problems

We conclude this section on graph algorithms with a brief mention of classic optimization problems defined on graphs. Standard references on combinatorial optimization include Vazirani (2001) and Korte and Vygen (2005). Given a weighted directed graph tt , classical combinatorial optimization problems include the following:

Minimum-weight spanning tree. A *minimum-weight spanning tree* of tt , denoted MST, is a spanning tree with the minimum possible weight. In order for the MST to exist, tt must contain a spanning tree. If all the weights of the individual edges are different, then the MST is unique.

Traveling salesperson problem. A *traveling salesperson tour* of tt , denoted TSP, is a cycle that passes through all the nodes of the digraph and has the minimum possible weight. In order for the TSP to exist, tt must contain a cycle through all nodes.

Multicenter optimization problems. Given a weighted digraph tt with vertices $V = \{v_1, \dots, v_n\}$ and a set $U = \{u_1, \dots, u_k\} \subset V$, the weighted distance from $v \in V$ to the set U is the smallest weighted distance from v to any vertex in $\{u_1, \dots, u_k\}$. We now consider the cost functions $H_{\max}, H_{\Sigma} : V^k \rightarrow \mathbb{R}$ defined by

$$H_{\max}(u_1, \dots, u_k) = \max_{i=1, \dots, k} \{ \min_{v \in V} \text{wdist}_{tt}(v, u_i) \}$$

$$\text{wdist}_{tt}(v_i, u_h),$$

$$i \in \{1, \dots, n\} \quad h \in \{1, \dots, k\}$$

$$\frac{n}{k}$$

$$H_{\Sigma}(u_1, \dots, u_k) = \min_{h \in \{1, \dots, k\}} \sum_{i=1}^n \text{wdist}_{tt}(v_i, u_h).$$

The *k-center problem* and the *k-median problem* consist of finding a set of vertices $\{u_1, \dots, u_k\}$ that minimizes the *k-center function* $H_{\max}$ and the *k-median function* H_{Σ} , respectively. We refer to [Vazirani\(2001\)](#)

for a discussion of the *k-center* and *k-median* problems (as well as the more general *uncapacited facility location* problem) over complete undirected graphs with edge costs satisfying the triangle inequality.

The Euclidean versions of these combinatorial optimization problems refer to the situation where one considers a weighted complete digraph whose vertex set is a point set in $\mathbb{R}^d$, $d \in \mathbb{N}$, and whose weight map assigns to each edge the Euclidean distance between the two nodes connected by the edge.

1.4.5 Algebraic graph theory

Algebraic graph theory (Biggs, 1994; Godsil and Royle, 2001) is the study of matrices defined by digraphs: in this section, we expose two topics. First, we study the equivalence between properties of graphs and of their associated adjacency matrices. We also specify how to associate a digraph to a nonnegative matrix. Second, we introduce and characterize the Laplacian matrix of a weighted digraph.

We begin by studying adjacency matrices. Note that the adjacency matrix of a weighted digraph is nonnegative and, in general, not stochastic. The following lemma expands on this point.

Lemma 1.31 (Weight-balanced digraphs and doubly stochastic adjacency matrices). *Let G be a weighted digraph of order n with weighted adjacency matrix A and weighted out-degree matrix D_{out} . Define the matrix*

$$F = \begin{cases} D_{\text{out}}^{-1} A, & \text{if each out-degree is strictly positive,} \\ (I_n + D_{\text{out}})^{-1} (I_n + A), & \text{otherwise.} \end{cases}$$

Then

- (i) F is row-stochastic; and
- (ii) F is doubly stochastic if G is weight-balanced and the weighted degree is constant for all vertices.

Proof. Consider first the case when each vertex has an outgoing edge so that D_{out} is invertible. We first note that

$\text{diag}(v)^{-1}v = 1_n$, for each $v \in (R \circ)^n$. Therefore

$$\sum_{\text{out}} 1 = \sum_{\text{in}} 1$$

which proves (i). Furthermore, if $D_{\text{out}} = D_{\text{in}} = dI_n$ for some $d \in$

$\mathbb{R}_{>0}$, then

$$D_{\text{out}}^{-1}A^T 1_n = \frac{1}{d} A^T 1_n = D_{\text{in}}^{-1}A^T 1_n = \text{diag}(A^T 1_n)^{-1} A^T 1_n = 1_n,$$

which proves (ii). Finally, if (V, E, A) does not have outgoing edges at each vertex, then apply the statement to the weighted digraph $(V, E \cup \{(i, i) \mid i \in$

$\{1, \dots, n\}, A + I_n)$. □

The next result characterizes the relationship between the adjacency matrix and directed paths in the digraph.

Lemma 1.32 (Directed paths and powers of the adjacency matrix). *Let t be a weighted digraph of order n with weighted adjacency matrix A , with unweighted adjacency matrix $A_{0,1} \in \{0,1\}^{n \times n}$, and possibly with self-*

loops. For all $i, j, k \in \{1, \dots, n\}$

- (i) the (i, j) entry of $A_{0,1}^k$ equals the number of directed paths of length k (including paths with self-loops) from node i to node j ; and*
- (ii) the (i, j) entry of A^k is positive if and only if there exists a directed path of length k (including paths with self-loops) from node i to node j .*

Proof. The second statement is a direct consequence of the first. The first statement is proved by induction. The statement is clearly true for $k = 1$.

Next, we assume the statement is true for $k - 1$ and we prove it for $k + 1$. By assumption, the entry $(A^k)_{ij}$ equals the number of directed paths from i to j of length k . Note that each path from i to j of length $k + 1$ identifies

(1) a unique node ℓ such that (i, ℓ) is an edge of t and (2) a unique path from ℓ to j of length k . We write $A^{k+1} = AA^k$ in components as

$$(A^{k+1})_{ij} = \sum_{\ell=1}^n A_{i\ell}(A^k)_{\ell j}.$$

Therefore, it is true that the entry $(A^{k+1})_{ij}$ equals the number of directed paths from i to j of length $k+1$. This concludes the induction argument. $\square$

The following proposition characterizes in detail the relationship between various connectivity properties of the digraph and algebraic properties of the adjacency matrix. The result is illustrated in Figure 1.13 and its proof is postponed until Section 1.8.3.

Proposition 1.33 (Connectivity properties of the digraph and positive powers of the adjacency matrix). *Let $\mathcal{G}$ be a weighted digraph of order n with weighted adjacency matrix A . The following statements are equivalent:*

- (i) $\mathcal{G}$ is strongly connected;
- (ii) A is irreducible; and
- (iii) $\sum_{k=0}^{n-1} A^k$ is positive.

For any $j \in \{1, \dots, n\}$, the following two statements are equivalent:

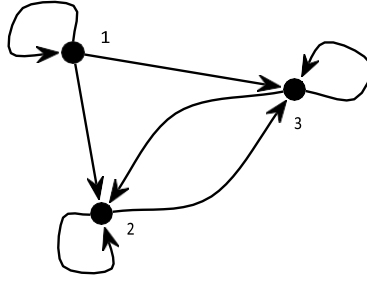


Figure 1.13 An illustration of Proposition 1.33. Even though vertices 2 and 3 are globally reachable, the digraph is not strongly connected because vertex 1 has no in-neighbor other than itself. Therefore, the associated adjacency matrix $A = (a_{ij})$ with $(a_{1j}) = 1_3$, $(a_{2j}) = (a_{3j}) = (0, 1, 1)$, is reducible.

(iv) the j th node of tt is globally reachable; and

(v) the j th column of $\sum_{k=0}^{n-1} A$ has positive entries.

Stronger statements can be given for digraphs with self-loops.

Proposition 1.34 (Connectivity properties of the digraph and positive powers of the adjacency matrix: cont'd). *Let tt be a weighted digraph of order n with weighted adjacency matrix A and with self-loops at each node. The following statements are equivalent:*

(iv) tt is strongly connected; and

(v) A^{n-1} is positive.

For any $j \in \{1, \dots, n\}$, the following two statements are equivalent:

(iv) the j th node of tt is globally reachable; and

(v) the j th column of A^{n-1} has positive entries.

Next, we characterize the relationship between irreducible aperiodic digraphs and primitive matrices (recall Definition 1.12). We will postpone the proof to Section 1.8.3.

Proposition 1.35 (Strongly connected and aperiodic digraph and primitive adjacency matrix). *Let tt be a weighted digraph of order n with weighted adjacency matrix A . The following two statements are equivalent:*

(i) tt is strongly connected and aperiodic; and

(ii) *A is primitive, that is, there exists $k \in \mathbf{N}$ such that A^k is positive.*

This concludes our study of adjacency matrices associated to weighted digraphs. Next, we emphasize how all results obtained so far have analogs that hold when the original object is a nonnegative matrix, instead of a weighted digraph.

Remark 1.36 (From a nonnegative matrix to its associated digraphs) $\times$ Given a nonnegative $n \times n$ matrix A , its *associated weighted digraph* is the weighted digraph with nodes $1, \dots, n$, and weighted adjacency matrix A . The unweighted version of this weighted digraph is called the *associated digraph*. The following statements are analogs of the previous lemmas:

- (i) if A is stochastic, then its associated digraph has weighted out-degree matrix equal to I_n ;
- (ii) if A is doubly stochastic, then its associated weighted digraph is weight-balanced and, additionally, both in-degree and out-degree matrices are equal to I_n ; and
- (iii) A is irreducible if and only if its associated weighted digraph is strongly connected. $\cdot$

So far, we have analyzed in detail the properties of adjacency matrices. We conclude this section by studying a second relevant matrix associated to a digraph, called the Laplacian matrix. The *Laplacian matrix* of the weighted digraph tt is

$$L(tt) = D_{\text{out}}(tt) - A(tt).$$

Some immediate consequences of this definition are the following:

- (i) $L(tt)1_n = 0_n$, that is, 0 is an eigenvalue of $L(tt)$ with eigenvector 1_n ;
- (ii) tt is undirected if and only if $L(tt)$ is symmetric; and
- (iii) $L(tt)$ equals the Laplacian matrix of the digraph obtained by adding to or removing from tt any self-loop with arbitrary weight.

Further properties are established as follows.

Theorem 1.37 (Properties of the Laplacian matrix). *Let $\mathcal{G}$ be a weighted digraph of order n . The following statements hold:*

- (i) all eigenvalues of $L(\mathcal{G})$ have nonnegative real part (thus, if $\mathcal{G}$ is undirected, then $L(\mathcal{G})$ is symmetric positive semidefinite);*
- (ii) if $\mathcal{G}$ is strongly connected, then $\text{rank}(L(\mathcal{G})) = n - 1$, that is, 0 is a simple eigenvalue of $L(\mathcal{G})$;*

- (iii) tt contains a globally reachable vertex if and only if $\text{rank}(L(tt)) = n - 1$;
- (iv) the following three statements are equivalent:
 - (a) tt is weight-balanced;
 - (b) $\frac{1}{n}^T L(tt) = \frac{0}{n}^T$; and
 - (c) $L(tt) + L(tt)^T$ is positive semidefinite.

1.5 DISTRIBUTED ALGORITHMS ON SYNCHRONOUS NETWORKS

Here, we introduce a synchronous network as a group of processors with the ability to exchange messages and perform local computations. What we present is a basic classic model studied extensively in the distributed algorithms literature. Our treatment is directly adopted with minor variations, from the texts by [Lynch \(1997\)](#) and [Peleg \(2000\)](#).

1.5.1 Physical components and computational models

Loosely speaking, a synchronous network is a group of processors, or nodes, that possess a local state, exchange messages along the edges of a digraph, and compute an update to their local state based on the received messages. Each processor alternates the two tasks of exchanging messages with its neighboring processors and of performing a computation step. Let us begin by describing what constitutes a network.

Definition 1.38 (Network). The physical component of a *synchronous network* S is a digraph (I, E_{cmm}) , where:

- (i) $I = \{1, \dots, n\}$ is called the *set of unique identifiers (UIDs)*; and
- (ii) E_{cmm} is a set of directed edges over the vertices $\{1, \dots, n\}$, called the *communication links*.

In general, the set of unique identifiers does not need to be n consecutive natural numbers, but we adopt this convention for simplicity. The set E_{cmm} models the topology of the communication service among the nodes: for $i, j \in \{1, \dots, n\}$, processor i can send a message to processor j if the directed edge (i, j) is present in E_{cmm} . Note that, unlike the standard treatments in [Lynch \(1997\)](#) and [Peleg \(2000\)](#), we do not assume the digraph to be strongly connected; the required connectivity assumption will be specified on a case-by-case basis.

Next, we discuss the state and the algorithms that each processor possesses and executes, respectively. By convention, we let the superscript $[i]$ denote any quantity associated with the node i .

Definition 1.39 (Distributed algorithm). A *distributed algorithm* **DA** for a network **S** consists of the sets

- (i) A , a set containing the null element, called the *communication alphabet*—elements of A are called *messages*;
- (ii) $W^{[i]}, i \in I$, called the *processor state sets*; and
- (iii) $W_0^{[i]} \subseteq W^{[i]}, i \in I$, sets of *allowable initial values*;

and of the maps

- (i) $\text{msg}^{[i]} : W^{[i]} \times I \rightarrow A, i \in I$, called *message-generation functions*;
and
- (ii) $\text{stf}^{[i]} : W^{[i]} \times A^n \rightarrow W^{[i]}, i \in I$, called *state-transition functions*.

If $W^{[i]} = W$, $\text{msg}^{[i]} = \text{msg}$, and $\text{stf}^{[i]} = \text{stf}$ for all $i \in I$, then **DA** is said to be *uniform* and is described by a tuple $(A, W, \{\text{msg}, \text{stf}\})$.

Now, with all elements in place, we can explain in more detail how a synchronous network executes a distributed algorithm (see Figure 1.14). The *state* of processor i is a variable $w^{[i]} \in W^{[i]}$, initially set equal to an

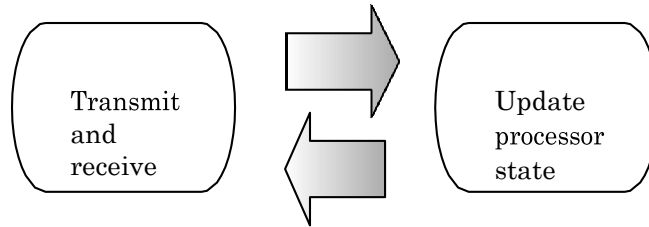


Figure 1.14 The execution of a distributed algorithm by a synchronous network.

allowable value $w_0^{[i]}$ in $W_0^{[i]}$. At each time instant $\ell \in \mathbb{Z}_{\geq 0}$, processor i sends to each of its out-neighbors j in the communication digraph (I, E_{cmm}) a message (possibly the null message) computed by

applying the message- generation function $\text{msg}^{[i]}$ to the current values of its state $w^{[i]}$ and to the identity j . Subsequently, but still at time instant $\ell \in \mathbb{Z}_{\geq 0}$, processor i updates the value of its state $w^{[i]}$ by applying the state-transition function $\text{stf}^{[i]}$ to the current value of $w^{[i]}$ and to the messages it receives from its in-neighbors. At each round, the first step is transmission and the second one is computation. These notions are formalized in the following definition.

Definition 1.40 (Network evolution). Let $\mathbf{DA}$ be a distributed algorithm for the network $\mathbf{S}$. The *evolution* of $(\mathbf{S}, \mathbf{DA})$ from initial conditions $w^{[i]} \in W^{[i]}$, $i \in I$, is the collection of trajectories $w^{[i]} : \mathbb{Z}_{\geq 0} \rightarrow W^{[i]}$, $i \in I$, satisfying

$$w^{[i]}(\ell) = \text{stf}^{[i]}(w^{[i]}(\ell - 1), y^{[i]}(\ell)),$$

where $w^{[i]}(-1) = w^{[i]}$, $i \in I$, and where the trajectory $y^{[i]} : \mathbb{Z}_{\geq 0} \rightarrow A^n$

(describing the messages received by processor i) has components $y^{[i]}(\ell)$, for

$j \in I$, given

$$\text{by } y^{[i]}(\ell) = \begin{cases} \text{msg}^{[j]}(w^{[j]}(\ell - 1), i), & \text{if } (j, i) \in E_{\text{cmm}}, \\ \text{null}, & \text{otherwise.} \end{cases}$$

Let $\ell \mapsto w(\ell) = (w^{[1]}(\ell), \dots, w^{[n]}(\ell))$ denote the collection of trajectories. •

We conclude this section with two sets of remarks. We first discuss some aspects of our communication model that have a large impact on the subsequent development. We then collect a few general comments about control structures and failure modes relevant in the study of distributed algorithms on networks.

Remarks 1.41 (Aspects of the communication model).

- (i) The network and the algorithm are referred to as *synchronous* because the communications between all processors takes place at the same time for all processors.
- (ii) Communication is modeled as a so-called “point-to-point” service: a processor can specify different messages for different out-neighbors and knows the processor identity corresponding to any incoming message.
- (iii) Information is exchanged between processors as messages, that is, elements of the alphabet A ; the message `null` indicates no communication. Messages might encode logical expressions such as `true`

and false, or finite-resolution quantized representations of integer and real numbers.

- (iv) In some uniform algorithms, the messages between processors are the processors' states. In such cases, the corresponding communication alphabet is $\mathbf{A} = W \cup \{\text{null}\}$ and the message-generation function $\text{msg}_{\text{std}}(w, j) = w$ is referred to as the *standard message-generation function*. .

Remarks 1.42 (Advanced topics: Control structures and failures).

- (i) Processors in a network have only partial information about the network topology. In general, each processor only knows its own UID, and the UID of its in- and out-neighbors. Sometimes we will assume that the processor knows the network diameter. In some cases (Peleg, 2000), actively running networks might depend upon “control structures,” that is, structures that are computed at initial time and are exploited in subsequent algorithms. For example, routing tables might be computed for routing problems, “leader” processors might be elected, and tree structures might be computed and represented in a distributed manner for various tasks; for example, coloring or maximal independent set problems. We present some sample algorithms to compute these structures below.
- (ii) A key issue in the study of distributed algorithms is the possible occurrence of failures. A network might experience intermittent or permanent communication failures: along given edges, a `null` message or an arbitrary message might be delivered instead of the intended value. Alternatively, a network might experience various types of processor failures: a processor might transmit only `null` messages (i.e., the `msg` function always returns `null`), a processor might quit updating its state (i.e., the `stf` function neglects incoming messages and returns the current state value), or a processor might implement arbitrarily modified `msg` and `stf` functions. The latter situation, in which completely arbitrary and possibly malicious behavior is adopted by faulty nodes, is referred to as a Byzantine failure in the distributed algorithms literature. •

1.5.2 Complexity notions

Here, we begin our analysis of the performance of distributed algorithms. We introduce a notion of algorithm completion and, in turn, we introduce the classic notions of time, space, and communication complexity.

Definition 1.43 (Algorithm completion). We say that an algorithm *terminates* when only `null` messages are transmitted and all processors’ states become constants.

•

Remarks 1.44 (Alternative termination notions).

- (i) In the interest of simplicity, we have defined evolutions to be unbounded in time and we do not explicitly require algorithms to

actually have termination conditions, that is, to be able to detect when termination takes place.

- (ii) It is also possible to define the termination time as the first instant when a given problem or task is achieved, independently of the fact that the algorithm might continue to transmit data subsequently. •

Defnition 1.45 (Time complexity). The *(worst-case) time complexity* of a distributed algorithm DA on a network S , denoted $TC(DA)$, is the maximum number of rounds required by the execution of DA on S among all allowable initial states until termination.

•

Next, it is of interest to quantify the memory and communication requirements of distributed algorithms. From an information theory view-point (Gallager, 1968), the information content of a memory variable or of a message is properly measured in bits. On the other hand, it is convenient to use the alternative notions of “basic memory unit” and “basic message.” It is customary (Peleg, 2000) to assume that a “basic memory unit” or a “basic message” contains $\log(n)$ bits; so that, for example, the information content of a robot identifier $i = 1, \dots, n$ is $\log(n)$ bits or, equivalently, one “basic memory unit.” Note that elements of the processor state set W or of the alphabet set A might amount to multiple basic memory units or basic messages; the null message has zero cost. Unless specified otherwise, the following definitions and examples are stated in terms of basic memory units and basic messages.

Defnition 1.46 (Space complexity). The *(worst-case) space complexity* of a distributed algorithm DA on a network S , denoted by $SC(DA, S)$, is the maximum number of

basic memory units required by a processor executing DA on S among all processors and among all allowable initial states until termination. .

Remark 1.47 (Space complexity conventions). By convention, each processor knows its identity, that is, it requires $\log(n)$ bits to represent its unique identifier in a set with n distinct elements. We do not count this cost in the space complexity of an algorithm.

.

Next, we introduce a notion of communication complexity.

Definition 1.48 (Communication complexity). The *(worst-case) communication complexity* of a distributed algorithm DA on a network S , denoted by $CC(DA)$, is the maximum number of basic messages transmitted over the entire network during the execution of DA among all allowable initial states until termination.

.

We conclude this section by discussing ways of quantifying time, space and communication complexity. The idea, borrowed from combinatorial optimization, is to adopt asymptotic “order of magnitude” measures. Formally, complexity bounds will be expressed with respect to the Bachmann–Landau symbols O , Ω and Θ defined in Section 1.1. Let us be more specific:

- (i) we will say that an algorithm has time complexity *of order* $\Omega(f(n))$ *over some network* if, for all n , there exists a network of order n and initial processor values such that the time complexity of the algorithm is greater than a constant factor times $f(n)$;
- (ii) we will say that an algorithm has time complexity *of order* $O(f(n))$ *over arbitrary networks* if, for all n , for all networks of order n and for all initial processor values, the time complexity of the algorithm is lower than a constant factor times $f(n)$; and
- (iii) we will say that an algorithm has time complexity *of order* $\Theta(f(n))$ if its time complexity is of order $\Omega(f(n))$ over some network and $O(f(n))$ over arbitrary networks at the same time.

Similar conventions will be used for space and communication complexity.

In many cases, the complexity of an algorithm will typically depend upon the number of vertices of the network. It is therefore useful to present a few simple facts about these functions now. Over arbitrary digraphs

$S = (I, E_{\text{cmm}})$ of order n , we have

$$\text{diam}(S) \in \Theta(n), \quad |E_{\text{cmm}}(S)| \in \Theta(n^2) \quad \text{and}$$

$$\text{radius}(v, S) \in \Theta(\text{diam}(S)), \text{ where } v \text{ is}$$

any vertex of S .

Remark 1.49 (Additional complexity notions). Numerous variations of the proposed complexity notions are possible and may be of interest.

Global lower bounds. In the definition of lower bound,

consider the logic quantifier describing the role of the network. The lower bound statement is “existential” rather than “global,” in the sense that the bound does not hold for all graphs. As discussed in Peleg (2000), it is possible to define also “global” lower bounds, that is, lower bounds over all graphs, or lower bounds over specified classes of graphs.

Average complexity notions. The proposed complexity notions focus on the worst-case situation. It is possible to define *expected* or *average* complexity notions, where one is interested in characterizing, for example, the average number of rounds required or the average number of basic messages transmitted over the entire network during the algorithm execution among all allowable initial states until termination.

Problem complexity. It is possible to define complexity notions for problems, rather than algorithms, by considering, for example, the worst-case optimal performance among all algorithms that solve the given problem, or over classes of algorithms or classes of graphs.

.

1.5.3 Broadcast and BFS tree computation

In the following, we consider some basic algorithmic problems such as the simple one-to-all communication task—that is, broadcasting—and the establishment of some “control structures” (see Remarks 1.42), such as the construction of a BFS spanning tree and the election of a leader.

Problem 1.50 (Broadcast). Assume that a processor, called the *source*, has a message, called the *token*. Transmit the token to all other processors in the network.

.

Note that existence of a spanning tree rooted at the source is a necessary requirement for the broadcast problem to be solvable. We begin by establishing some analysis results for the broadcast problem.

Lemma 1.51 (Complexity lower bounds for the broadcast problem). *Let S be a network containing a spanning tree rooted at v . The broadcast problem for S from the source v has communication complexity lower bounded by $n - 1$ and time complexity lower bounded by $\text{radius}(v, S)$.*

In what follows, we shall solve the broadcast problem while simultaneously also considering the following

problem.

Problem 1.52 (BFS tree computation). Let G be a network containing a spanning tree rooted at v . Compute a distributed representation for a BFS tree rooted at v .

.

We add two remarks on the BFS tree computation problem:

- (i) By a distributed representation of a directed tree with bounded memory at each node, we mean the following: each child vertex knows the identity of its parent and the root vertex knows that it has no parents. A more informative structure would require each parent to know the identity of its children; this is easy to achieve on undirected digraphs.
- (ii) The BFS tree computation has the same lower bounds as the broadcast problem.

An elegant and classic solution to the broadcast and BFS tree computation problems is given by the flooding algorithm. This algorithm implements the same “breadth-first search” mechanism of the (centralized) BFS algorithm characterized in Lemma 1.28:

[Informal description] The source broadcasts the token to its out-neighbors. In each communication round, each node determines whether it has received a non-null message from one of its in-neighbors. When a non-null message is received—that is, the token is received—the node performs two actions. First, the node stores the token in the variable *data* (this solves the Broadcast problem). Second, the node stores the identity of one of the transmitting in-neighbors in the variable *parent* (this solves the BFS tree computation problem). Specifically, if the message is received simultaneously from multiple in-neighbors, then the node stores the smallest among the identities of the transmitting in-neighbors. In the subsequent communication round, the node broadcasts the token to its out-neighbors.

To formally describe the algorithm, we assume that the node with the message to be broadcast is $v = 1$. Also, we assume that the token is a letter of the Greek alphabet $\{\alpha, \dots, \omega\}$:

Synchronous Network: $\mathbf{S} = (\{1, \dots, n\}, E_{\text{cmm}})$

Distributed Algorithm: flooding

Alphabet: $\mathbf{A} = \{\alpha, \dots, \omega\} \cup \text{null}$

Processor State: $w = (\text{parent}, \text{data}, \text{snd-flag})$, where

```
function msg(w, i)
1: if (parent = i) AND (snd-flag = true) then
2:   return data
3: else
4:   return null
function stf(w, y)
```

```

1: case
2:   (data = null) AND (y contains only null messages):
      % The node has not yet received the token
3:     new-parent := null
4:     new-data := null
5:     new-snd-flag := false
6:   (data = null) AND (y contains a non-null message):
      % The node has just received the token
7:     new-parent := smallest UID among transmitting in-
neighbors
8:     new-data := a non-null message
9:     new-snd-flag := true
10:  (data = null):
      % If the node already has the token, then do not re-broadcast it
11:    new-parent := parent
12:    new-data := data
13:    new-snd-flag := false
14: return (new-parent, new-data, new-snd-flag)

```

An execution of the flooding algorithm is shown in Figure 1.15.

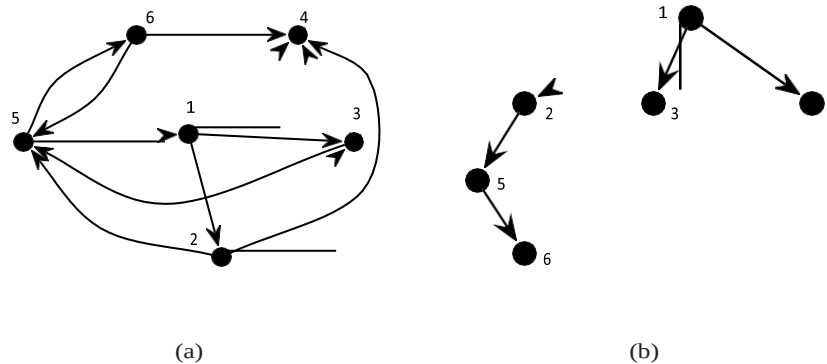


Figure 1.15 An example execution of the flooding algorithm. The source is vertex 1: (a) shows the network and (b) shows the BFS tree that results from the execution.

This algorithm can be analyzed by induction: one can show that, for d $\in \{1, \dots, \text{radius}(v, \cdot)\}$, every node at a distance d from the root receives a non-null message at round d . A

summary of the results is given as follows.

Lemma 1.53 (Complexity upper bounds for the flooding algorithm). *For a network S containing a spanning tree rooted at v , the flooding algorithm has communication complexity in $\Theta(|E_{\text{cmm}}|)$, time complexity in $\Theta(\text{radius}(v, S))$, and space complexity in $\Theta(1)$.*

We conclude the section with a final remark.

Remark 1.54 (Termination condition for the flooding algorithm). As presented, the flooding algorithm does not include a termination condition, that is, the processors do not have a mechanism to detect when the broadcast and tree computation are complete. If an upper bound on the graph diameter is known, then it is easy to design a termination condition based on this information; we do this in the next subsection. If no *a priori* knowledge is available, then one can design more sophisticated algorithms for networks with stronger connectivity properties. We refer to [Lynch \(1997\)](#) and [Peleg \(2000\)](#) for a complete discussion about this.

•

1.5.4 Leader election

Next, we formulate another interesting problem for a network.

Problem 1.55 (Leader election). Assume that all processors of a network have a state variable, say `leader`, initially set to `unknown`. We say that a leader is elected when one and only one processor has the state variable set to `true` and all others have it set to `false`. Elect a leader.

•

This task is a bit more global in nature. We display here a solution that requires individual processors to know the diameter of the network, denoted by $\text{diam}(S)$, or an upper bound on it:

[Informal description] In each communication round, each agent sends to its out-neighbors the maximum UID it has received up to that time. This is repeated for $\text{diam}(S)$ rounds. At the last

round, each agent compares the maximum received UID with its own, and declares itself a leader if they coincide, or a non-leader otherwise.

The algorithm is called the `floodmax` algorithm: the maximum UID in the network is transmitted to other agents in an incremental fashion. At the first communication round, agents that are neighbors of the agent with the maximum UID receive the message from it. At the next communication round, the neighbors of these agents receive the message with the maximum UID. This process goes on for $\text{diam}(\)$ rounds, to ensure that every agent receives the maximum UID. Note that there are networks for which all agents receive the message with the maximum UID in fewer communication rounds than $\text{diam}(S)$. The algorithm is formally stated as follows:

Synchronous Network: $S = (\{1, \dots, n\}, E_{\text{cmm}})$

Distributed Algorithm: `floodmax`

Alphabet: $\mathbf{A} = \{1, \dots, n\} \cup \{\text{null}\}$

Processor State: $w = (\text{my-id}, \text{max-id}, \text{leader}, \text{round})$, where

$\text{my-id} \in \{1, \dots, n\}$, initially: $\text{my-id}^{[i]} = i$ for all i
 $\text{max-id} \in \{1, \dots, n\}$, initially: $\text{max-id}^{[i]} = i$ for all i
 $\text{leader} \in \{\text{false}, \text{true}, \text{unknown}\}$, initially: $\text{leader}^{[i]} = \text{unknown}$ for all i
 $\text{round} \in \{0, 1, \dots, \text{diam}(\mathbf{S})\}$, initially: $\text{round}^{[i]} = 0$ for all i

function $\text{msg}(w, i)$

```

1: if round < diam( ) then
2:   return max-id
3: else
4:   return null

```

function $\text{stf}(w, y)$

```

1: new-id := max{max-id, largest identifier in y}
2: case
3:   round < diam( ): new-lead := unknown
4:   round = diam( ) AND max-id = my-id: new-lead := true
5:   round =

```

```

diam( ) AND max-id > my-id: new-lead := false
6: return (my-id, new-id, new-lead, round + 1)

```

Figure 1.16 shows an execution of the floodmax algorithm. Some properties of this algorithm are characterized in the following lemma. A complete analysis of this algorithm, including modifications to improve the communication complexity, is discussed in Lynch (1997, Section 4.1).

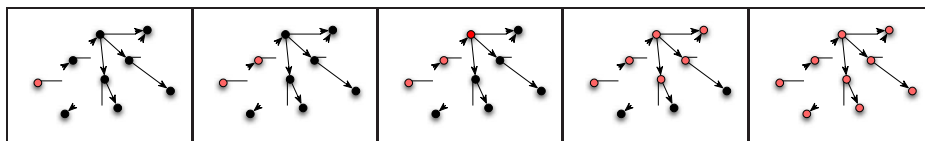


Figure 1.16 Execution of the floodmax algorithm. The diameter of the network is 4. In the leftmost frame, the agent with the maximum UID is colored in red. After four communication rounds, its message has been received by all agents.

Lemma 1.56 (Complexity upper bounds for the floodmax algorithm). *For a network S containing a spanning tree, the floodmax algorithm has communication complexity in $O(\text{diam}(S)|E_{\text{cmm}}|)$, time complexity equal to $\text{diam}(S)$, and space complexity in $\Theta(1)$.*

A simplification of the floodmax algorithm leads to the Le Lann–Chang–Roberts algorithm (or LCR algorithm in short) for leader election in rings, see (Lynch, 1997, Chapter 3.3), which we describe next. The

LCR algorithm runs on a ring digraph and does not require the agents to know the diameter of the network. We provide an informal and a formal description of the algorithm.

[Informal description] In each communication round, each agent sends to its neighbors the maximum UID it has received up to that time. (Agents do not record the number of communication rounds.) When the agent with the maximum UID receives its own UID from an in-neighbor, it declares itself the leader.

Synchronous Network: **ring**

digraph Distributed

Algorithm: LCR Alphabet:

$\mathbf{A} = \{1, \dots, n\} \cup \{\text{null}\}$

Processor State: $w = (\text{my-id}, \text{max-id}, \text{leader}, \text{snd-flag})$,
where

$\text{my-id} \in \{1, \dots, n\},$	initially: $\text{my-id}^{[i]} = i$ for all i
$\text{max-id} \in \{1, \dots, n\},$	initially: $\text{max-id}^{[i]} = i$ for
all i leader	$\in \{\text{true}, \text{false}, \text{unknown}\},$
initially: $\text{leader}^{[i]} = \text{unknown}$ for all i	$\text{snd-flag} \in$
$\{\text{true}, \text{false}\},$	initially: $\text{snd-flag}^{[i]} =$
true for all i	

function $\text{msg}(w, i)$

```

1: if  $\text{snd-flag} = \text{true}$  then
2:   return  $\text{max-id}$ 
3: else
4:   return null

```

function $\text{stf}(w, y)$

```

1: case

```

```

2:  (y contains only null msgs) OR (largest identifier in y <
   my-id):
3:    new-id := max-id
4:    new-lead := leader
5:    new-snd-flag := false
6:  (largest identifier in y = my-id):
7:    new-id := max-id
8:    new-lead := true
9:    new-snd-flag := false
10: (largest identifier in y > my-id):
11:   new-id := largest identifier in y
12:   new-lead := false
13:   new-snd-flag := true
14: return (my-id, new-id, new-lead, new-snd-flag)

```

Figure 1.17 shows an execution of the LCR algorithm. The properties of the LCR algorithm can be characterized as follows.

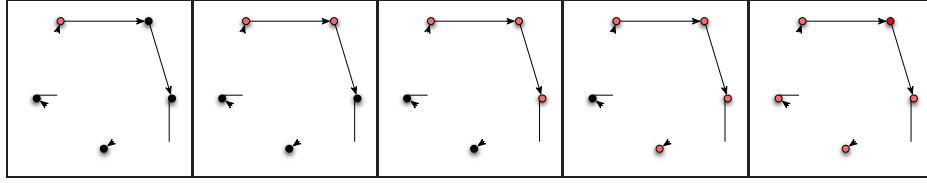


Figure 1.17 Execution of the LCR algorithm. In the leftmost frame, the agent with the maximum UID is colored in red. After five communication rounds, this agent receives its own UID from its in-neighbor and declares itself the leader.

Lemma 1.57 (Complexity upper bounds for the LCR algorithm). *For a ring network $\mathcal{S}$ of order n , the LCR algorithm has communication complexity in $\Theta(n^2)$, time complexity equal to n , and space complexity in $\Theta(1)$.*

1.5.5 Shortest-paths tree computation

Finally, we consider the shortest-paths tree problem in a weighted digraph: in Section 1.4.4 we presented the Dijkstra algorithm to solve this problem in a centralized setting; we present here the Bellman-Ford algorithm for the distributed setting. We consider a synchronous network associated to a weighted digraph, that is, we assume that a strictly positive weight is associated to each communication edge. We aim to compute a tree containing shortest paths from a source, say node 1, to all other nodes. As for the computation of a BFS tree, we aim to obtain a distributed representation of a directed tree with bounded memory at each node:

[Informal description] Each agent maintains in its memory an estimate dist of its weighted distance from the source, and an estimate parent of the in-neighbor corresponding to the (weighted) shortest path from the source. The

`dist` estimate is initialized to 0 for the source and to $+$ for all other nodes. In each communication round, each agent performs the following tasks: (1) it transmits its `dist` value estimate to its out-neighbors, (2) it computes the smallest quantity among “the `dist` value received from an in-neighbor summed with the edge weight corresponding to that in-neighbor,” and (3) if the agent’s estimate `dist` is larger than this quantity, then the agent updates its `dist` and its estimate parent.

The algorithm is formally stated as follows:

Synchronous Network with Weights: $\mathbf{S} = (\{1, \dots, n\}, E_{\text{cmm}}, \mathbf{A})$ Distributed Algorithm:

Distributed Bellman-Ford Alphabet: $\mathbf{A} = \mathbf{R}_{>0}$

$\cup \text{null} \cup \{+\infty\}$

Processor State: $w = (\text{parent}, \text{dist})$, where

$\text{parent} \in \{1, \dots, n\}, \quad \text{initially: } \text{parent}^{[j]} = j \text{ for all } j$

$\text{dist} \in \mathbf{A}, \quad \text{initially: } \text{data}^{[1]} = 0, \\ \text{data}^{[j]} = +\infty \text{ for all } j \neq 1$

function $\text{msg}(w, i)$

1: if $\text{round} < n$ then

2: return dist

3: else

4: return null

function $\text{stf}(w, y)$

1: $i := \text{processor UID}$

2: $k := \arg \inf_j \{y_j + a_{ji} \mid \text{all } y_j = \text{null}\}$

3: if $(\text{dist} < k)$ then

4: return $(\text{parent}, \text{dist})$

5: else

6: return $(k, y_k + a_{ki})$

In other words, if we let $d_i \in \mathbf{R}_{\geq 0} +$ denote the dist variable for each processor i , then the Bellman-Ford algorithm is equivalent to the following discrete-time dynamical system:

$$d_i(\ell + 1) = \inf \{d_i(\ell)\}, \inf \{d_j(\ell) + a_{ji} \mid (j, i) \in E_{\text{cmm}}\},$$

with initial conditions $d(0) = (1, +\infty, \dots, +\infty)$. (Recall

that E_{cmm} is the edge set and that the weights a_{ij} are strictly positive for all $(i, j) \in E_{\text{cmm}}$.)

The following formal statements may be made about the evolution of this algorithm. If there exists a directed spanning tree rooted at vertex 1, then all variables d_i will take a final value in time equal to their topological distance from vertex 1. After k communication rounds, the estimated distance at node i equals the shortest path of topological length at most k from the source to node i . Therefore, after $n - 1$ communication rounds, all possible distinct topological paths connecting source to node i have been investigated.

The complexity properties of the distributed Bellman-Ford algorithm are described as follows.

Lemma 1.58 (Complexity upper bounds for the distributed Bellman-Ford algorithm). *For a network G of order n containing a spanning tree rooted at v , the distributed Bellman-Ford algorithm has communication complexity in $\Theta(|E_{\text{cmn}}|)$, time complexity equal to $n - 1$, and space complexity in $\Theta(1)$.*

Figure 1.18 shows an execution of the distributed Bellman-Ford algorithm in a weighted digraph with four nodes and six edges.

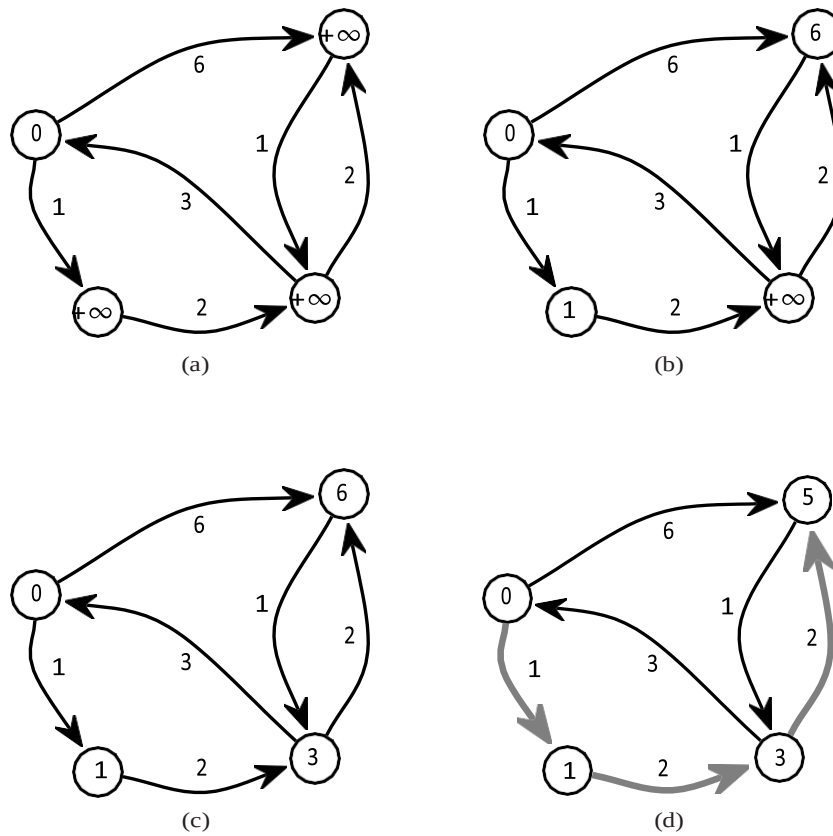


Figure 1.18 Execution of the distributed Bellman-Ford algorithm. (a) The processor state initialization. The vertex 1 is the only one whose variable `dist` is 0. After three iterations, as guaranteed by Lemma 1.58, (d) depicts the resulting shortest-paths tree of the digraph rooted at vertex 1. This tree is represented in the last frame, with edges colored in gray.

1.6 LINEAR DISTRIBUTED ALGORITHMS

Computing a linear combination of the initial states of the processors is one of the most basic computation that we might be interested in implementing on a synchronous network. More accurately, linear distributed algorithms on synchronous networks are discrete-time linear dynamical systems whose evolution map is linear and has a sparsity structure related to the network. These algorithms represent an important class of iterative algorithms that find applications in optimization, in the solution of systems of equations, and in distributed decision making; see, for instance [Bertsekas and Tsitsiklis \(1997\)](#). In this section, we present some relevant results on distributed linear algorithms.

1.6.1 Linear iterations on synchronous networks

Given a synchronous network $S = (\{1, \dots, n\}, E_{\text{cmm}})$, assign a scalar f_{ji} to each directed edge $(i, j) \in E_{\text{cmm}}$. Given such scalars f_{ji} for $(i, j) \in E_{\text{cmm}}$, the Linear combination algorithm over S is defined as follows:

Distributed Algorithm: Linear combination

Alphabet: $\mathbf{A} = \mathbf{R} \cup$

null Processor

state: $w \in \mathbf{R}$

function $\text{msg}(w, i) = \text{msg}_{\text{std}}(w, i)$

function $\text{stf}(w, y)$

1: $i := \text{processor UID}$

2: return $f_{ii}w + \sum_{j \in \mathbf{N}^{\text{in}}(i)} f_{ij}y_j$

We assume that each processor $i \in \{1, \dots, n\}$ knows the scalars f_{ij} , for $j \in N^{\text{in}}(i)$, so that it can evaluate the state-transition function. Also, we assume that real numbers may be transmitted through a communication channel, that is, we neglect quantization issues in the message-generation function.

In the language of Section 1.3, one can regard the Linear combination algorithm over $\mathbb{R}$ as the discrete-time continuous-space dynamical system (X, X_0, f) , with $X = X_0 = \mathbb{R}^n$ and an evolution map defined by $f(w) = F \cdot w$, where we define a matrix $F \in \mathbb{R}^{n \times n}$ with vanishing entries except for f_{ji} , for $(i, j) \in E_{\text{cmm}}$. Note that, if $A(S)$ denotes the adjacency matrix of the digraph S , then the entries of F vanish precisely when the entries of

$A(S)^T$ vanish. With this notation, the evolution $w : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ with initial condition $w_0 \in \mathbb{R}^n$ is given by

$$w(0) = w_0, \quad w(\ell + 1) = F \cdot w(\ell), \quad \ell \in \mathbb{Z}_{\geq 0}. \quad (1.6.1)$$

Conversely, any linear algorithm of the form (1.6.1) can easily be cast as a Linear combination algorithm over a suitable synchronous network. We do this bookkeeping carefully, in order to be consistent with the notion of associated weighted digraph from Remark 1.36. Given $F \in \mathbb{R}^{n \times n}$, let

S_F be the synchronous network with node set $\{1, \dots, n\}$ and with edge set $E_{\text{cmm}}(F)$, defined by any of the equivalent statements:

- (i) $(i, j) \in E_{\text{cmm}}(F)$ if and only if $f_{ji} \neq 0$; or
- (ii) S_F is the reversed and unweighted version of the digraph associated to F .

1.6.2 Averaging algorithms

Here, we study linear combination algorithms over time-dependent weighted directed graphs; we restrict our analysis to nonnegative weights.

Definition 1.59 (Averaging algorithms). The *averaging algorithm* associated to a sequence of stochastic matrices $F(\ell) \in \mathbb{R}^{n \times n}$ is the discrete-time dynamical system

$$w(\ell + 1) = F(\ell) \cdot w(\ell), \quad \ell \in \mathbb{Z}_{\geq 0}. \quad (1.6.2)$$

In the literature, such algorithms are often referred to as agreement algorithms, or as consensus algorithms.

There are useful ways to compute a stochastic matrix, and therefore, a time-independent averaging algorithm, from a weighted digraph; see Exercise E1.15.

Definition 1.60 (Adjacency- and Laplacian-based averaging). Let G be a weighted digraph with node set $\{1, \dots, n\}$, weighted adjacency matrix A , weighted out-degree matrix D_{out} , and weighted Laplacian L . Then

- (i) the *adjacency-based averaging algorithm* is defined by the stochastic matrix $(I_n + D_{\text{out}})^{-1}(I_n + A)$ and reads in components

$$w_i(\ell + 1) = \frac{1}{\text{out}(i)} w_i(\ell) + \sum_{j=1}^n a_{ij} w_j(\ell) ; \quad (1.6.3)$$

(ii) given a positive scalar ε upper bounded by $\min\{1/d_{\text{out}}(i) \mid i \in \{1, \dots, n\}\}$, the *Laplacian-based averaging algorithm* is defined by the stochastic matrix $I_n - \varepsilon L(tt)$ and reads in components

$$w_i(\ell + 1) = (1 - \varepsilon) w_i(\ell) + \varepsilon \sum_{j=1, j \neq i}^n a_{ij} w_j(\ell). \quad (1.6.4)$$

These notions are immediately extended to sequences of stochastic matrices arising from sequences of weighted digraphs.

Adjacency-based averaging algorithms arising from unweighted undirected graphs without self-loops are also known as *equal-neighbor averaging rule* or the *Vicsek's model* (see [Vicsek et al., 1995](#)). Specifically, if tt is an unweighted graph with vertices $1, \dots, n$ and without self-loops, then the equal-neighbor averaging rule is

$$w_i(\ell + 1) = \text{avrg} \left[\sum_{j \in N_{tt}(i)} \{w_j(\ell)\} \right], \quad (1.6.5)$$

where we adopt the shorthand $\text{avrg}(\{x_1, \dots, x_k\}) = (x_1 + \dots + x_k)/k$.

Remark 1.61 (Sensing versus communication interpretation of directed edges). In the definition of averaging algorithms arising from digraphs, the digraph edges play the role of “sensing edges,” not that of “communication edges.” In other words, a nonzero entry a_{ij} , corresponding to the digraph edge (i, j) , implies that the i th component of the state is updated with the j th component of the state. It is as if node i could sense the state of node j , rather than node i transmitting to node j its own state.

Next, we present the main stability and convergence

results for averaging algorithms associated to a sequence of stochastic matrices. We start by discussing equilibrium points and their stability. Recall that $\mathbf{1}_n$ is an eigenvector of any stochastic matrix with eigenvalue 1 and that the diagonal set $\text{diag}(\mathbb{R}^n)$ is the vector subspace generated by $\mathbf{1}_n$. Therefore, any point in $\text{diag}(\mathbb{R}^n)$ is an equilibrium for any averaging algorithm. We refer to the points of the $\text{diag}(\mathbb{R}^n)$ as *agreement configurations*, since all the components of an element in $\text{diag}(\mathbb{R}^n)$ are equal to the same value. We will informally say that an algorithm *achieves agreement* if it steers the network state toward the set of agreement configurations.

Lemma 1.62 (Stability of agreement configurations). *Any averaging algorithm in $\mathbb{R}^n$ is uniformly stable and uniformly bounded with respect to $\text{diag}(\mathbb{R}^n)$.*

Regarding convergence results, we need to introduce a useful property of collections of stochastic matrices. Given $\alpha \in]0, 1]$, the set of *non-degenerate matrices with respect to α* consists of all stochastic matrices F with entries

f_{ij} , for $i, j \in \{1, \dots, n\}$, satisfying

$$f_{ii} \in [\alpha, 1], \quad \text{and} \quad f_{ij} \in \{0\} \cup [\alpha, 1] \text{ for } j \neq i.$$

Additionally, the sequence of stochastic matrices $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$ is *non-degenerate* if there exists $\alpha \in]0, 1]$ such that $F(\ell)$ is non-degenerate with respect to α for all $\ell \in \mathbb{Z}_{\geq 0}$. We now state the main convergence result and postpone its proof to Section 1.8.5.

Theorem 1.63 (Convergence for time-dependent stochastic matrices). *Let $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\} \subset \mathbb{R}^{n \times n}$ be a non-degenerate sequence of stochastic matrices. For $\ell \in \mathbb{Z}_{\geq 0}$ let $tt(\ell)$ be the unweighted digraph associated to*

$F(\ell)$, according to Remark 1.36. The following statements are equivalent:

- (i) *the set $\text{diag}(\mathbb{R}^n)$ is uniformly globally attractive for the averaging algorithm associated to $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$; and*
- (ii) *there exists a duration $\delta \in \mathbb{N}$ such that, for all $\ell \in \mathbb{Z}_{\geq 0}$, the digraph*

$$tt(\ell + 1) \cup \dots \cup tt(\ell + \delta)$$

contains a globally reachable vertex.

We collect a few observations about this result.

Remarks 1.64 (Discussion of Theorem 1.63).

- (i) The statement in Theorem 1.63(i) means that each solution to the time-dependent linear dynamical system (1.6.2) converges uniformly and asymptotically to the vector subspace generated by $\mathbf{1}_n$.
- (ii) The necessary and sufficient condition in Theorem 1.63(ii) amounts to the existence of a uniformly bounded time duration δ with the property that a weak connectivity assumption holds over each collection of δ consecutive digraphs. We refer to Blondel et al. (2005) for a counterexample showing that if the duration in Theorem 1.63 is not uniformly bounded, then there exist algorithms that do not converge.
- (iii) According to Definition 1.23, uniform convergence is a property of all solutions to system (1.6.2) starting at *any arbitrary time*, and not

only at time equal to zero. If we restrict our attention to solutions that only start at time zero, then Theorem 1.63 should be modified as follows: the statement in Theorem 1.63(i) implies, but is not implied by, the statement in Theorem 1.63(ii).

- (iv) The theorem applies only to sequences of non-degenerate matrices. Indeed, there exist sequences of degenerate stochastic matrices whose associated averaging algorithms converge. Furthermore, one does not even need to consider sequences, because it is possible to define converging algorithms by just considering a single stochastic matrix. Precisely when the stochastic matrix is primitive, we already know that the associated averaging algorithm will converge (see Theorem 1.13). Examples of degenerate primitive stochastic matrices (with converging associated averaging algorithms) are given in Exercise E1.23. We discuss time-invariant averaging algorithms in Proposition 1.68 below. •

Theorem 1.63 gives a general result about non-degenerate stochastic matrices that are not necessarily symmetric. The following theorem presents a convergence result for the case of symmetric matrices (i.e., undirected digraphs) under connectivity requirements that are weaker (i.e., the duration does not need to be uniformly bounded) than those expressed in statement (ii) of Theorem 1.63.

Theorem 1.65 (Convergence for time-dependent stochastic symmetric matrices). *Let $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\} \subset \mathbb{R}^{n \times n}$ be a non-degenerate sequence of symmetric, stochastic matrices. For $\ell \in \mathbb{Z}_{\geq 0}$, let $tt(\ell)$ be the un-*

weighted graph associated to $F(\ell)$, according to Remark 1.36. The following statements are equivalent:

- (i) *the set $\text{diag}(\mathbb{R}^n)$ is globally attractive for the averaging algorithm associated to $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$; and*
- (ii) *for all $\ell \in \mathbb{Z}_{\geq 0}$, the graph*

$$\begin{matrix} \blacksquare & tt(\tau) \\ \tau \geq \ell \end{matrix}$$

is connected.

Let us particularize our discussion here on adjacency- and Laplacian-based averaging algorithms.

Corollary 1.66 (Convergence of adjacency- and Laplacian-based averaging algorithms). *Let $tt(\ell) \in \mathbb{R}^{n \times n}$ be a sequence of weighted digraphs. The following statements are equivalent:*

- (i) *there exists $\delta \in \mathbf{N}$ such that, for all $\ell \in \mathbf{Z}_{\geq 0}$, the digraph*

$$tt(\ell + 1) \cup \cdots \cup tt(\ell + \delta)$$

contains a globally reachable vertex;
- (ii) *the set $\text{diag}(\mathbf{R}^n)$ is uniformly globally attractive for the adjacency-*

based averaging algorithm (1.6.3) associated to $\{tt(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$;

and

- (iii) the set $\text{diag}(\mathbb{R}^n)$ is uniformly globally attractive for the Laplacian-based averaging algorithm (1.6.4) (defined with $\varepsilon < 1/n$) associated to $\{tt(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$.

Finally, we refine the results presented thus far by discussing some further aspects.

Proposition 1.67 (Convergence to a point in the invariant set). *Under the assumptions in Theorem 1.63 and assuming that $\text{diag}(\mathbb{R}^n)$ is uniformly globally attractive for the averaging algorithm, each individual evolution converges to a specific point of $\text{diag}(\mathbb{R}^n)$.*

In general, the final value upon which all $w_i, i = 1, \dots, n$, agree in the limit is unknown. This final value depends on the initial condition and the specific sequence of matrices defining the time-dependent linear algorithm. In some cases, however, one can compute the final value by restricting the class of allowable matrices. We consider two settings: time-independent averaging algorithms and doubly stochastic averaging algorithms.

First, we specialize the main convergence result to the case of time-independent averaging algorithms. Note that, given a stochastic matrix F , convergence of the averaging algorithm associated to F for all initial conditions is equivalent to the matrix F being semi-convergent (see Definition 1.6).

Proposition 1.68 (Time-independent averaging algorithm). *Consider the linear dynamical system on $\mathbb{R}^n$*

$$w(\ell + 1) = Fw(\ell), \quad \ell \in \mathbb{Z}_{\geq 0}. \quad (1.6.6)$$

Assume that $F \in \mathbb{R}^{n \times n}$ is stochastic, let $tt(F)$ denote its associated weighted digraph, and let $v \in \mathbb{R}^n$ be a left eigenvector of F with eigenvalue 1. Assume either one of the two following properties:

- (i) F is primitive (i.e., $tt(F)$ is strongly connected and aperiodic); or

(ii) F has non-zero diagonal terms and a column of F^{n-1} has positive entries (i.e., $\text{tt}(F)$ has self-loops at each node and has a globally reachable node).

Then every trajectory w of system (1.6.6) converges to $(v^T w(0)/v^T 1_n)1_n$.

Proof. From Theorem 1.63 we know that the dynamical system (1.6.6) converges if property (ii) holds. The same conclusion follows if F satisfies prop-

erty (i) because of the Perron–Frobenius Theorem 1.13 and Lemma 1.7. To computing the limiting value, note that

$$v^T w(\ell + 1) = v^T F w(\ell) = v^T w(\ell),$$

that is, the quantity $\ell \mapsto v^T w(\ell)$ is constant. Because F is semi-convergent and stochastic, we know that $\lim_{\ell \rightarrow +\infty} w(\ell) = \alpha 1_n$ for some α . To conclude, we compute α from the relationship $\alpha(v^T 1_n) = \lim_{\ell \rightarrow +\infty} v^T w(\ell) = v^T w(0)$. $\square$

Remarks 1.69 (Alternative conditions for time-independent averaging).

- (i) The following necessary and sufficient condition generalizes and is weaker than the two sufficient conditions given in Proposition 1.68: every trajectory of system (1.6.6) is asymptotically convergent if and only if all sinks of the condensation digraph of $tt(F)$ are aperiodic subgraphs of $tt(F)$. We refer the interested reader to Meyer (2001, Chapter 8) for the proof of this statement and for the related notion of ergodic classes of a Markov chain. Also, we refer the interested reader to Exercise E1.13 for the notion of condensation digraph.
- (ii) Without introducing any trajectory w , the result of the proposition can be equivalently stated by saying that

$$\lim_{\ell \rightarrow +\infty} F^\ell = (v^T 1_n)^{-1} 1_n v^T.$$

Second, we focus on the case of doubly stochastic averaging algorithms.

Corollary 1.70 (Average consensus). *Let $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$ be a sequence of stochastic matrices as in Theorem 1.63. If all matrices $F(\ell)$, $\ell \in \mathbb{Z}_{\geq 0}$, are doubly stochastic, then every trajectory w of the averaging algorithms satisfies*

$$\sum_{i=1}^n w_i(\ell) = \sum_{i=1}^n w_i(0), \quad \text{for all } \ell,$$

that is, the sum of the initial conditions is a conserved quantity. Therefore, if $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$ is non-degenerate and satisfies property (ii) in Theorem 1.63, then

$$\lim_{l \rightarrow +\infty} w_j(\ell) = \prod_{i=1}^n w_i(0), \quad j \in \{1, \dots, n\}.$$

Proof. The proof of the first fact is an immediate consequence of

$$\sum_{i=1}^n w_i(\ell + 1) = \mathbf{1}^T \mathbf{W}(\ell + 1) = \mathbf{1}^T \mathbf{F}_H(\ell) \mathbf{w}(\ell) = \mathbf{1}^T \mathbf{W}(\ell) = \sum_{i=1}^n w_i(\ell).$$

The second fact is an immediate consequence of the first fact. $\square$

In other words, if the matrices are doubly stochastic, then each component of the trajectories will converge to the average of the initial condition. We therefore adopt the following definition: an *average-consensus averaging algorithm* is an averaging algorithm whose sequence of stochastic matrices are all doubly stochastic.

1.6.3 The convergence speed of averaging algorithms

We know that any trajectory of the associated averaging algorithm converges to the diagonal set $\text{diag}(\mathbf{R}^n)$; in what follows we characterize how fast this convergence takes place. We begin with some general definitions for semi-convergent matrices (recall the discussion culminating in Lemma 1.7).

Definition 1.71 (Convergence time and exponential convergence factor). Let $A \in \mathbf{R}^{n \times n}$ be semi-convergent with limit $\lim_{l \rightarrow +\infty} A^l = A^*$.

(i) For $\varepsilon \in]0, 1[$, the ε -convergence time of A is the smallest time

$T_\varepsilon(A) \in \mathbf{Z}_{\geq 0}$ such that, for all $x_0 \in \mathbf{R}^n$ and $\ell \geq T_\varepsilon(A)$,

$$\|A^\ell x_0 - A^* x_0\|_2 \leq \varepsilon \|x_0 - A^* x_0\|_2.$$

(ii) The *exponential convergence factor* of A , denoted by $r_{\text{exp}}(A) \in [0, 1[$, is

$$\|A^\ell x_0 - A^* x_0\|_2^{1/\ell}$$

$$\lim_{l \rightarrow +\infty} \|A^l x_0 - A^* x_0\|_2^{1/l}$$

$$r_{\text{exp}}(A) = \sup_{x_0} \limsup_{\ell \rightarrow \infty} \frac{\|x(\ell) - x^*\|}{\|x_0 - x^*\|}.$$

The exponential convergence factor has the following interpretation: If the trajectory $x(\ell) = A^\ell x_0$ maximizing the sup operator has the form $x(\ell) = \rho^\ell(x_0 - x^*) + x^*$, for $\rho < 1$, then it is immediate to see that $r_{\text{exp}}(A) = \rho$.

Lemma 1.72 (Exponential convergence factor of a convergent matrix). *If A is a convergent matrix, then $r_{\text{exp}}(A) = \rho(A)$.*

In what follows, we are interested in studying how the convergence time

and the exponential convergence factor of a matrix depend upon ε and upon the dimension of the matrix itself.

Remark 1.73 (Complexity notions). Analogously to the treatment in Section 1.5.2, we introduce some complexity notions. Let $A_n \in \mathbb{R}^{n \times n}$, $n \in \mathbb{N}$, be a sequence of semi-convergent matrices with limit $\lim_{l \rightarrow +\infty} A^l = A_n^*$, and let $\varepsilon \in]0, 1]$. We say that:

- (i) $T_\varepsilon(A_n)$ is of order $\Omega(f(n, \varepsilon))$ if, for all n and all ε , there exists an initial condition $x_0 \in \mathbb{R}^n$ such that $\|A^l x_0 - A^* x_0\|_2 > \varepsilon$ for all times ℓ greater than a constant factor times $f(n, \varepsilon)$;

- (ii) $T_\varepsilon(A_n)$ is of order $O(f(n, \varepsilon))$ if, for all n and all ε , $T_\varepsilon(A_n)$ is less than or equal to a constant factor times $f(n, \varepsilon)$; and

- (iii) $T_\varepsilon(A_n)$ is of order $\Theta(f(n, \varepsilon))$ if it is both of order $\Omega(f(n, \varepsilon))$ and of order $O(f(n, \varepsilon))$.

Lemma 1.74 (Asymptotic relationship). Let $A_n \in \mathbb{R}^{n \times n}$, $n \in \mathbb{N}$, be a sequence of semi-convergent matrices and let $\varepsilon \in]0, 1]$. In the limit as $\varepsilon \rightarrow 0^+$ and as $n \rightarrow +\infty$,

$$T_\varepsilon(A_n) = O\left(\frac{1}{1 - r_{\exp}(A_n)} \sum_{n \in \mathbb{N}} \log \varepsilon^{-1}\right).$$

Proof. By the definition of the exponential convergence factor and of lim sup, we know that for all $\eta > 0$, there exists N such that, for all $\ell > N$,

$$\|A^\ell x_0 - A^* x_0\|_2 \leq (r_{\exp}(A_n) + \eta)^\ell \|x_0 - A^* x_0\|_2.$$

The ε -convergence time is upper bounded by any ℓ such that $(r_{\exp}(A_n) + \eta)^\ell \leq \varepsilon$. Selecting $\eta = (1 - r_{\exp}(A_n))/2$, simple manipulations lead to

$$\ell \geq \frac{1}{-\log((r_{\exp}(A_n) + 1)/2)} \log \varepsilon^{-1}.$$

It is also immediate to note that $\frac{2}{1-r} \geq \frac{1}{-\log((r+1)/2)}$, for all $r \in]0, 1[$.
This

establishes the bound in the statement above. $\square$

Next, we apply the notion of convergence time and exponential convergence factor to any non-degenerate stochastic matrix whose associated digraph has a globally reachable node.

Lemma 1.75 (Exponential convergence factor of stochastic matrices).
Let F be a stochastic matrix with strictly positive diagonal entries and whose associated digraph has a globally reachable node. Then

$$r_{\exp}(F) = \rho_{\text{ess}}(F).$$

(From equation (1.2.1), recall that $\rho_{\text{ess}}(F) = \max\{|\lambda| \mid \lambda \in \text{spec}(F) \setminus \{1\}\}$.)

Proof. If $v \in \mathbb{R}^n$ is a left eigenvector of F , then, as in Proposition 1.68,

$$\lim_{l \rightarrow +\infty} F^l = F^* = (v^T 1_n)^{-1} 1_n v^T.$$

Relying upon $v^T F = v^T$ and $F 1_n = 1_n$, straightforward manipulations show that $F^* = F^* F = F F^* = F^* F^*$ and in turn

$$F^{l+1} - F^* = (F - F^*)(F^l - F^*).$$

For any $w_0 \in \mathbb{R}^n$ such that $w_0 = F^* w_0$, define the error variable $e(\ell) := F^l w_0 - F^* w_0$. Note that the error variable evolves according to $e(\ell + 1) = (F - F^*)e(\ell)$ and converges to zero. Additionally, the rate at which $w(\ell) = F^l w_0$ converges to $F^* w_0$ is the same at which $e(\ell)$ converges to zero, that is,

$$r_{\text{exp}}(F - F^*) = r_{\text{exp}}(F).$$

Therefore,

$$r_{\text{exp}}(F) = r_{\text{exp}}(F - F^*) = \rho(F - F^*) = \rho_{\text{ess}}(F).$$

□

The following result establishes bounds on convergence factors and convergence times for stochastic matrices arising from the equal-neighbor averaging rule in equation (1.6.5).

Theorem 1.76 (Bounds on the convergence factor and the convergence time). *Let G be an undirected unweighted connected graph of order n and let $\varepsilon \in]0, 1]$. Define the stochastic matrix $F = (I_n + D(G))^{-1}(I_n + A(G))$. There exists $\gamma > 0$ (independent of n) such that the exponential convergence factor and convergence time of F satisfy*

$$r_{\text{exp}}(F) \leq 1 - \gamma n^{-3}, \quad \text{and} \quad T_{\varepsilon}(F) \in O(n^3 \log \varepsilon^{-1}),$$

as $\varepsilon \rightarrow 0^+$ and $n \rightarrow +\infty$.

1.6.4 Algorithms defined by tridiagonal Toeplitz and tridiagonal circulant matrices

This section presents a detailed analysis of the convergence rates of linear distributed algorithms defined by tridiagonal Toeplitz matrices and by certain circulant matrices. Let us start by introducing the family of matrices

under study. For $n \geq 2$ and $a, b, c \in \mathbb{R}$, define the $n \times n$ matrices $\text{Trid}_n(a, b, c)$ and $\text{Circ}_n(a, b, c)$ by

$$\text{Trid}_n(a, b, c) = \begin{pmatrix} b & c & 0 & \dots & 0 \\ a & b & c & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & a & b & c \\ 0 & \dots & 0 & a & b \end{pmatrix},$$

and

$$\text{Circ}_n(a, b, c) = \text{Trid}_n(a, b, c) + \begin{pmatrix} 0 & \dots & \dots & 0 & a \\ 0 & \dots & \dots & 0 & c \end{pmatrix}.$$

We call the matrices Trid_n and Circ_n *tridiagonal Toeplitz* and *tridiagonal circulant*, respectively. The two matrices only differ in their $(1, n)$ and $(n, 1)$

entries. Note our convention that

$$\text{Circ}_n(a, b, c) = \begin{pmatrix} b & a+c \\ a+c & b \end{pmatrix}.$$

Note that, for $a = 0$ and $c = 0$ (alternatively, $a = 0$ and $c = 0$), the synchronous networks defined by $\text{Trid}(a, b, c)$ and $\text{Circ}(a, b, c)$ are, respectively, the chain and the ring digraphs introduced in Section 1.4. If both a and c are non-vanishing, then the synchronous networks are, respectively, the undirected versions of the chain and the ring digraphs.

Now, we characterize the eigenvalues and eigenvectors of Trid_n and Circ_n .

Lemma 1.77 (Eigenvalues and eigenvectors of tridiagonal Toeplitz and tridiagonal circulant matrices). *For $n \geq 2$ and $a, b, c \in \mathbb{R}$, the following statements hold:*

(i) for $ac \neq 0$, the eigenvalues and eigenvectors of $\text{Trid}_n(a, b, c)$ are, respectively, for $i \in \{1, \dots, n\}$,

$$\lambda_i = \frac{a}{c} \cos \frac{i\pi}{n+1} \in \mathbb{C}, \quad v_i = \begin{pmatrix} \frac{a}{c} \sum_{j=1}^{n/2} \sin \frac{j\pi}{n+1} \\ \frac{a}{c} \sum_{j=2/2}^{n/2} \sin \frac{2j\pi}{n+1} \\ \vdots \\ \frac{a}{c} \sum_{j=n/2}^{n/2} \sin \frac{ni\pi}{n+1} \end{pmatrix} \in \mathbb{C}^{n+1};$$

(ii) the eigenvalues and eigenvectors of $\text{Circ}_n(a, b, c)$ are, respectively,

$$\text{for } i \in \{1, \dots, n\} \text{ and } \omega = \exp\left(\frac{2\pi i}{n}\right),$$

$$b + (a + c) \cos \frac{i2\pi}{n} + \sqrt[n]{-1}(c - a) \sin \frac{i2\pi}{n} \in \mathbb{C},$$

$$\text{and } (1, \omega^i, \dots, \omega^{(n-1)i})^T \in \mathbb{C}^n.$$

Proof. Both facts are discussed, for example, in Meyer (2001, Example 7.2.5 and Exercise 7.2.20). Fact (ii) requires some straightforward algebraic manipulations.

□

Figure 1.19 illustrates the location of the eigenvalues of these matrices in the complex plane.

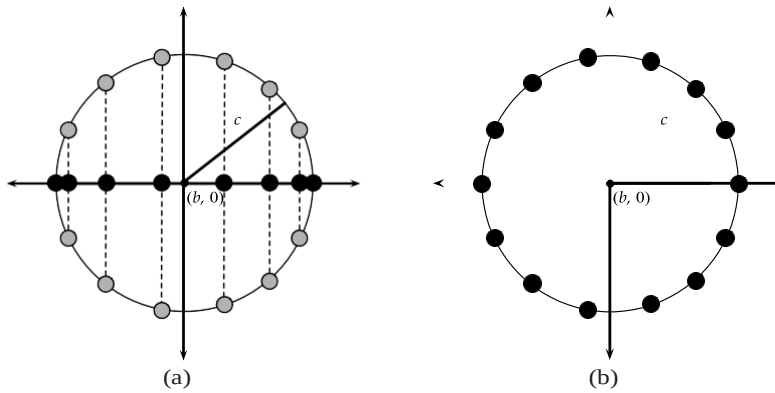


Figure 1.19 The eigenvalues of Toeplitz and circulant matrices (cf., Lemma 1.77) are closely related to the roots of unity. Plotted in the complex plane, the black disks correspond in (a) to the eigenvalues of $\text{Trid}_{13}(a, b, c)$, and in (b) to the eigenvalues of $\text{Circ}_{14}(0, b, c)$.

Remarks 1.78 (Inclusion relationships for eigenvalues of tridiagonal Toeplitz and tridiagonal circulant matrices).

- (i) The set of eigenvalues of $\text{Trid}_n(a, b, c)$ is contained in the real interval

complex plane $[b - 2\sqrt{-1} |ac|, b + 2\sqrt{-1} |ac|]$, if $ac \geq 0$, and in the interval in the complex plane $[b - 2\sqrt{-1} |ac|, b + 2\sqrt{-1} |ac|]$, if $ac \leq 0$.

- (ii) The set of eigenvalues of $\text{Circ}_n(a, b, c)$ is contained in the ellipse on the complex plane with center b , horizontal axis $2|a+c|$, and vertical axis $2|c-a|$.

Next, we characterize the convergence rate of linear algorithms defined by tridiagonal Toeplitz and tridiagonal circulant matrices. As in the previous section, we are interested in asymptotic results as the system dimension $n \rightarrow +\infty$ and as the accuracy parameter ε goes to 0^+ .

Theorem 1.79 (Linear algorithms defined by tridiagonal Toeplitz and tridiagonal circulant matrices). *Let $n \geq 2$, $\varepsilon \in]0, 1[$, and $a, b, c \in \mathbb{R}$. Let $x: \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ and $y: \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ be solutions to*

$$x(\ell + 1) = \text{Trid}_n(a, b, c)x(\ell), \quad y(\ell + 1) = \text{Circ}_n(a, b, c)y(\ell),$$

with initial conditions $x(0) = x_0$ and $y(0) = y_0$, respectively. The following statements hold:

- (i) *if $a = c = 0$ and $0 < |b| < 1$, then $\lim_{\ell \rightarrow +\infty} x(\ell) = 0_n$ with ε -convergence time in $\Theta(n \log \varepsilon^{-1})$;*
- (ii) *if $a \neq 0$, $c = 0$ and $0 < |b| < 1$, then $\lim_{\ell \rightarrow +\infty} x(\ell) = 0_n$ with ε -convergence time in $\Theta(n \log n + \log \varepsilon^{-1})$; and*
- (iii) *if $q \geq 0, 1 - q \geq 0, 1 \geq b \geq 0$ and $a + b + c = 1$, then $\lim_{\ell \rightarrow +\infty} y(\ell) = \frac{1}{n} \mathbf{1}_n y_0$ with ε -convergence time in $\Theta(n \log \varepsilon^{-1})$.*

The proof of this result is reported in Section 1.8.6. Next, we extend these results to another interesting set of tridiagonal matrices. For $n \geq 2$ and $a, b \in \mathbb{R}$, define the $n \times n$ matrices $\text{ATrid}^+(a, b)$ and $\text{ATrid}^-(a, b)$ by

$$\text{ATrid}_{\vec{n}}^{\pm}(a, b) = \text{Trid}_n(a, b, a) \pm \begin{bmatrix} a & 0 & \dots & \dots & 0 \\ 0 & 0 & \dots & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & 0 \\ 0 & \dots & \dots & 0 & a \end{bmatrix}$$

We refer to these matrices as *augmented tridiagonal* matrices. If we define $\mathbf{1}_n$

$$\mathbf{1}_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \end{bmatrix}$$

$$P_+ = \begin{pmatrix} 1 & -1 & 1 & 0 & \dots & 0 \\ 1 & 0 & -1 & 1 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 1 & 0 & \dots & 0 & -1 & 1 \\ \vdots & & & & & \vdots \\ 1 & 0 & \dots & 0 & 0 & -1 \end{pmatrix}$$

and

$$P = \begin{pmatrix} 1 & 1 & 0 & 0 & \dots & 0 \\ -1 & 0 & 1 & 1 & \dots & 0 \\ & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & & & & (-1)^{n-2} & 0 & \dots & 0 & 1 & 1 \end{pmatrix},$$

then the following similarity transforms are satisfied:

$$\text{ATrid}^\pm(a, b) = P_\pm \begin{pmatrix} b \pm 2a & 0 \\ 0 & \text{Trid}_{n-1}(a, b, a) \end{pmatrix} P_\pm^{-1}. \quad (1.6.7)$$

To analyze the convergence properties of the linear algorithms determined by $\text{ATrid}^+(a, b)$ and $\text{ATrid}^-(a, b)$, we will find it useful to consider the vector

$$1_{n-}^T = (1, -1, 1, \dots, (-1)^{n-2}, (-1)^{n-1})^T \in \mathbb{R}^n.$$

In the following theorem, we will not assume that the matrices of interest are semi-convergent. We will establish convergence to a trajectory, rather than to a fixed point. For $\varepsilon \in]0, 1[$, we say that a trajectory $x : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$

converges to $x_{\text{final}} : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ with convergence time $T_\varepsilon \in \mathbb{Z}_{\geq 0}$ if

- (i) $\|x(\ell) - x_{\text{final}}(\ell)\|_2 \rightarrow 0$ as $\ell \rightarrow +\infty$; and
- (ii) T_ε is the smallest time such that $\|x(\ell) - x_{\text{final}}(\ell)\|_2 \leq \varepsilon \|x(0) - x_{\text{final}}(0)\|_2$, for all $\ell \geq T_\varepsilon$.

Theorem 1.80 (Linear algorithms defined by augmented tridiagonal matrices). *Let $n \geq 2$, $\varepsilon \in]0, 1[$, and $a, b \in \mathbb{R}$ with $a \neq 0$ and*

$|b| + 2|a| = 1$. Let $x : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ and $z : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ be solutions to

$$x(\ell + 1) = \text{ATrid}^+(a, b)x(\ell), \quad z(\ell + 1) = \text{ATrid}^-(a, b)z(\ell),$$

with initial conditions $x(0) = x_0$ and $z(0) = z_0$, respectively. The following statements hold:

- (i) $\lim_{l \rightarrow +\infty} \|x(l) - x_{\text{ave}}(l)\|_n = 0_n$, where $x_{\text{ave}}(l) = (1_n x_0)(b + 2a)$,
(ii) $\lim_{l \rightarrow +\infty} \|z(l) - z_{\text{ave}}(l)\|_n = 0_n$, where $z_{\text{ave}}(l) = (1_n^T z_0)(b - 2a)$, with ε -convergence time in $\Theta(n^2 \log \varepsilon^{-1})$.

The proof of this result is reported in Section 1.8.6.

Remark 1.81 (From Toeplitz to stochastic matrices). A tridiagonal Toeplitz matrix is not stochastic unless its off-diagonal elements are zero. The tridiagonal circulant matrices Circ_n and augmented tridiagonal matrices ATrid_n^+ studied in Theorem 1.79(iii) and Theorem 1.80(i) are slight

modifications of tridiagonal Toeplitz matrices and are doubly stochastic. In- deed, the evolutions converge to the average consensus value, as predicted by Corollary 1.70. Note that convergence times obtained for Circ_n and ATrid^+ are consistent with the upper bound predicted by Theorem 1.76.

•

We conclude this section with some useful bounds.

Lemma 1.82 (Bounds on vector norms). Assume that $x \in \mathbb{R}^n$, $y \in$

$\mathbb{R}^{n-1}$, and $z \in \mathbb{R}^{n-1}$ jointly satisfy

$$x = P_+ \begin{matrix} \Sigma \\ 0 \\ y \end{matrix}, \quad x = P_- \begin{matrix} \Sigma \\ 0 \\ z \end{matrix}.$$

Then $\frac{1}{2} \|x\|_2 \leq \|y\|_2 \leq (n-1) \|x\|_2$ and $\frac{1}{2} \|x\|_2 \leq \|z\|_2 \leq (n-1) \|x\|_2$.

The proof of this result is based on spelling out the coordinate expressions for x , y , and z , and is left to the reader as Exercise E1.29.

1.7 NOTES

Dynamical systems and stability theory

Our definition of a state machine is very basic; more general definitions of state machines can be found in the literature (see Sipser, 2005), but the one presented in this chapter is sufficient for our purposes.

The literature on dynamical and control systems is vast. The main tool that we use in later chapters is the LaSalle Invariance Principle, obtained by LaSalle (1960) and discussed in LaSalle (1986); see also the earlier works by Barbašin and Krasovskiĭ (1952) and Krasovskiĭ (1963)

for related versions. Relevant sample references include modern texts on dynamical systems ([Guckenheimer and Holmes, 1990](#)), linear control systems ([Chen, 1984](#)), nonlinear control systems ([Khalil, 2002](#)), robust control ([Dullerud and Paganini, 2000](#)), and discrete-event systems ([Cassandras and Lafortune, 2007](#)).

Graph theory

The basic definitions of graph theory are standard in the literature; see, for example, [Biggs \(1994\)](#), [Godsil and Royle \(2001\)](#), and [Diestel \(2005\)](#). The discussion about graph algorithms is taken from [Cormen et al. \(2001\)](#), which also contains detailed discussion on implementation and complexity issues.

Regarding Section 1.4.4.4, standard references on combinatorial optimization include Vazirani (2001) and Korte and Vygen (2005).

In Section 1.4.5, all statements about powers of the adjacency matrix are standard results in algebraic graph theory; see, for example Biggs (1994) and Godsil and Royle (2001). Lemma 1.27 is a recent result from Lin et al. (2005) and Moreau (2005). Proposition 1.35, on the fact that a weighted digraph is aperiodic and irreducible if and only if its adjacency matrix is primitive, is related to standard results in the theory of Markov chains; see, for example Seneta (1981) and Meyn and Tweedie (1999). Our proof adopts the approach in Lin (2005). Laplacian matrices have numerous remarkable properties; two elegant surveys are Mohar (1991) and Merris (1994). Theorem 1.37, characterizing the properties of the Laplacian matrix, contains some recent results. A proof of statement (ii) is given in Olfati-Saber and Murray (2004); in our proof, we follow the approach in Francis (2006). Statement (iii) is proved by Lin et al. (2005) and Francis (2006); the following equivalent version is proved in Ren and Beard (2005): a weighted digraph tt contains a spanning tree if and only if $\text{rank}(L(\text{rev}(tt))) = n - 1$. Regarding statement (iv), the equivalence between (iv)a and (iv)b is proved by Olfati-Saber and Murray (2004) and the equivalence between (iv)b and (iv)c is proved by Moreau (2005).

Distributed algorithms

Our discussion of distributed algorithms is extremely incomplete. We have only presented a few token ideas and we refer to the textbooks by Lynch (1997) and Peleg

(2000) for detailed treatments. Let us mention briefly that many more efficient algorithms are available in the literature—for example, the GHS algorithm (Gallager et al., 1983) for minimum spanning tree computation and consensus algorithms with communication and processors faults; much attention is dedicated to fault tolerance in asynchronous systems with shared memory and in asynchronous network systems.

Linear distributed algorithms

Distributed linear algorithms—and, in particular, averaging iterations that achieve consensus among processors—have a long and rich history. The richness comes from the vivid analogies with physical processes of diffusion, with Markov chain models, and with the sharp theory of positive matrices developed by Perron and Frobenius. What follows is a necessarily incomplete list. An early reference on averaging opinions and achieving consensus

is [DeGroot \(1974\)](#). An early reference on the connection between averaging algorithms, the products of stochastic matrices, and ergodicity in inhomogeneous Markov chains is [Chatterjee and Seneta \(1977\)](#) – the history of inhomogeneous Markov chains being a classic topic since the early twentieth century. The stochastic setting was investigated in [Cogburn \(1984\)](#). Load balancing with divisible tasks in parallel computers is discussed in [Cybenko \(1989\)](#). A comprehensive theory of asynchronous parallel processors implementing distributed gradient methods and time-dependent averaging algorithms is developed in the series of works [Tsitsiklis \(1984\)](#), [Tsitsiklis et al. \(1986\)](#), and [Bertsekas and Tsitsiklis \(1997\)](#). Much interest for averaging algorithms arose from the influential work on flocking by [Jadbabaie et al. \(2003\)](#). Sharp conditions for convergence for the time-dependent setting were obtained in [Moreau \(2005\)](#). Finally, proper attention was given to the average consensus problem in [Olfati-Saber and Murray \(2004\)](#).

Regarding Theorem [1.63](#), characterizing the convergence of averaging algorithms defined by sequences of stochastic matrices, we note that: (1) the PhD thesis [Tsitsiklis \(1984\)](#) established convergence under a strong-connectivity assumption; (2) a sufficient condition was independently re-discovered in [Jadbabaie et al. \(2003\)](#), adopting a result from [Wolfowitz \(1963\)](#); and (3) [Moreau \(2003, 2005\)](#) obtained the necessary and sufficient condition (for uniform convergence in non-degenerate sequences) involving the existence of a uniformly globally reachable node. The work in [Moreau \(2003, 2005\)](#) is an early reference also for Theorem [1.65](#); additional related results and a historical discussion appeared in [Blondel et al. \(2005\)](#) and [Hendrickx \(2008\)](#). The estimates of the convergence factor given in Theorem [1.76](#) in Section [1.6.3](#) were proved by [Landau and Odlyzko](#)

(1981). Our treatment in Section 1.6.4 follows Martínez et al. (2007a).

Among the numerous recent directions of research on consensus and averaging, we would like to mention the following: continuous-time consensus algorithms (Olfati-Saber and Murray, 2004; Moreau, 2004; Lin et al., 2004; Ren and Beard, 2005; Lin et al., 2005, 2007c), consensus over random networks (Hatano and Mesbahi, 2005; Wu, 2006; Patterson et al., 2007; Picci and Taylor, 2007; Porfiri and Stilwell, 2007; Tahbaz-Salehi and Jadbabaie, 2008; Fagnani and Zampieri, 2009), consensus in finite time (Cortés, 2006; Sundaram and Hadjicostis, 2008), consensus in small-world networks (Olfati-Saber, 2005; Durrett, 2006; Tahbaz-Salehi and Jadbabaie, 2007), consensus algorithms for general functions (Bauso et al., 2006; Cortés, 2008b; Lorenz and Lorenz, 2008; Sundaram and Hadjicostis, 2008), connections with the heat equation and partial difference equation (Ferrari-Trecate et al., 2006), spatially decaying interactions (Cucker and Smale, 2007), convergence in time-delayed and asynchronous settings (Blondel et al., 2005; Angeli and Bliman, 2006; Fang and Antsaklis, 2008), quantized consensus

problems (Savkin, 2004; Kashyap et al., 2007; Carli et al., 2009; Zhu and Martínez, 2008b), consensus on manifolds (Scardovi et al., 2007; Sarlette and Sepulchre, 2009; Igarashi et al., 2007), applications to distributed signal processing (Spanos et al., 2005; Xiao et al., 2005; Olfati-Saber et al., 2006; Zhu and Martínez, 2008a), characterization of convergence rates and time complexity (Landau and Odlyzko, 1981; Olshevsky and Tsitsiklis, 2009; Cao et al., 2008; Carli et al., 2008). Numerous interesting results are reported in recent PhD theses (Lin, 2005; Cao, 2007; Lorenz, 2007; Barooah, 2007; Carli, 2008; Hendrickx, 2008; Sarlette, 2009). Finally, we would like to point out two recent surveys (Olfati-Saber et al., 2007; Ren et al., 2007) and the text by (Ren and Beard, 2008).

Synchronization is a fascinating topic related to averaging algorithms. A very early reference is the work by Huygens (1673) on coupled pendula. The synchronization of oscillators in dynamical systems has received increasing attention, and key references include Wiener (1958), Kuramoto (1975), Winfree (1980), Kuramoto (1984), Strogatz (2000), and Nijmeijer (2001); see also the widely accessible Strogatz (2003). Under all-to-all interactions, Mirollo and Strogatz (1990) prove synchronization of a collection of “integrate and fire” biological oscillators. Recent works on the Kuramoto and other synchronized oscillator models include Jadbabaie et al. (2004), Chopra and Spong (2009), Triplett et al. (2006), Papachristodoulou and Jadbabaie (2006), Wang and Slotine (2006).

1.8 PROOFS

This section gathers the proofs of the main results presented in the chapter.

1.8.1 Proof of Theorem 1.21

Here we provide the proof of the LaSalle Invariance Principle for set-valued discrete-time dynamical systems. We remark that Theorem 1.19 is an immediate consequence of Theorem 1.21 and that Theorem 1.20 is proved in a similar way (for details, we refer to (Khalil, 2002)).

Proof of Theorem 1.21. Let γ be any evolution of (X, X_0, T) starting from W . Let $\Omega(\gamma)$ denote the ω -limit set⁶ of the sequence $\gamma = \{\gamma(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$; since W is closed, it follows that $\Omega(\gamma) \subset W$. Next, we prove that $\Omega(\gamma)$ is weakly positively invariant. Let $z \in \Omega(\gamma)$. Then there exists a subsequence

⁶The ω -limit set of a sequence $\gamma = \{\gamma(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$ is the set of points γ for which there exists a subsequence $\{\gamma(l_m) \mid m \in \mathbb{Z}_{\geq 0}\}$ of γ such that $\lim_{m \rightarrow +\infty} \gamma(l_m) = \gamma$.

$\{\gamma(\ell_m) \mid m \in \mathbb{Z}_{\geq 0}\}$ of γ such that $\lim_{m \rightarrow +\infty} \gamma(\ell_m) = z$. Consider the sequence

$\{\gamma(\ell_m + 1) \mid m \in \mathbb{Z}_{\geq 0}\}$. Since this sequence is bounded, it has a convergent subsequence. For ease of notation, we use the same notation to refer to it, that is, there exists y such that $\lim_{m \rightarrow +\infty} \gamma(\ell_m + 1) = y$. By definition,

$y \in \Omega(\gamma)$. Moreover, using the fact that T is closed, we deduce that $y \in T(z)$. Therefore, $\Omega(\gamma)$ is weakly positively invariant.

Now, consider the sequence $V \circ \gamma = \{V(\gamma(\ell)) \mid \ell \in \mathbb{Z}_{\geq 0}\}$. Since γ is bounded and V is non-increasing along T on W , the sequence $V \circ \gamma$ is decreasing and bounded from below, and therefore, convergent. Let $c \in \mathbb{R}$ satisfy $\lim_{\ell \rightarrow +\infty} V(\gamma(\ell)) = c$. Next, we prove that the value of V on $\Omega(\gamma)$ is

constant and equal to c . Take any $z \in \Omega(\gamma)$. Accordingly, there exists a subsequence $\{\gamma(\ell_m) \mid m \in \mathbb{Z}_{\geq 0}\}$ such that $\lim_{m \rightarrow +\infty} \gamma(\ell_m) = z$. Since V is continuous, $\lim_{m \rightarrow +\infty} V(\gamma(\ell_m)) = V(z)$. From $\lim_{\ell \rightarrow +\infty} V(\gamma(\ell)) = c$, we conclude

$$\lim_{m \rightarrow +\infty} V(\gamma(\ell_m)) = V(z) = c.$$

Finally, the fact that $\Omega(\gamma)$ is weakly positively invariant and V being constant on $\Omega(\gamma)$ implies that

$$\Omega(\gamma) \subset \{x \in X \mid \exists y \in T(x) \text{ such that } V(y) = V(x)\}.$$

Therefore, we conclude that $\lim_{\ell \rightarrow +\infty} \text{dist}(\gamma(\ell), S \cap V^{-1}(c)) = 0$, where S is the largest weakly positively invariant set contained in $\{x \in X \mid \exists y \in T(x) \text{ such that } V(y) = V(x)\}$. $\square$

Proof of Lemma 1.26. The first statement is obvious. Regarding the second statement, we prove that a topologically balanced digraph with a globally reachable node is strongly connected, and leave the proof of the other case to the reader. We reason by contradiction. Assume that tt is not strongly connected. Let $S \subseteq V$ be the set of all nodes of tt that are globally reachable. By hypothesis, $S \neq \emptyset$. Since tt is not strongly connected, we have $S \subsetneq V$. Note that any outgoing edge with origin in a globally reachable node automatically makes the destination a globally reachable node too. This implies that there cannot be any outgoing edges from a node in S to a node in $V \setminus S$. Let $v \in V \setminus S$ such that v has an out-neighbor in S (such a node must exist, since otherwise the nodes in S cannot be globally reachable). Since by hypothesis tt is balanced, there must exist an edge of the form $(w, v) \in E$. Clearly, $w \in S$, since otherwise v would be globally reachable too, which is a contradiction. Therefore, $w \in S$. Again, using the

fact that tt is topologically balanced, there must exist an edge of the form $(z, w) \in E$. As before, $z \in V \setminus S$ (note that $z = v$ is a possibility). Since $V \setminus S$ is finite and so is the number of possible edges between its nodes, applying this argument repeatedly, we find that there exists a vertex whose out-degree is strictly larger than its in-degree, which is a contradiction with the fact that tt is topologically balanced. We refer to Cortés (2008b) for the proof that tt is Eulerian.

□

Proof of Lemma 1.27. (i) $\Rightarrow$ (ii) Assume that $i \in V$ is the root of the spanning tree and take an arbitrary pair of nonempty, disjoint sub-sets $U_1, U_2 \subset V$. If $i \in U_1$, then there must exist a path from $i \in U_1$ to a

node in U_2 . Therefore, U_2 must have an in-neighbor. Analogously, if $i \in U_2$, then U_1 must have an in-neighbor. Finally, it is possible that $i \notin U_1 \cup U_2$. In this case, there exist paths from i to both U_1 and U_2 , that is, both sets have in-neighbors.

(ii) $\Rightarrow$ (i) This is proved by finding a node from which there exists a path to all others. We do this in an algorithmic manner using induction. At each induction step k , except the last one, four sets of nodes are considered, $U_1(k) \subset W_1(k) \subset V$, $U_2(k) \subset W_2(k) \subset V$, with the following properties:

- (a) the sets $W_1(k)$ and $W_2(k)$ are disjoint; and
- (b) from each node of $U_s(k)$ there exists a path to each other node in $W_s(k) \setminus U_s(k)$, $s \in \{1, 2\}$.

Induction Step $k=1$: Set $U_1 = W_1 = i_1$ and $U_2 = W_2 = i_2$, where i_1, i_2 are two arbitrary different nodes of the graph that satisfy the properties (a) and (b).

Induction Step $k > 1$: Suppose that for $k - 1$ we found

sets $U_1(k-1) \subset W_1(k-1)$ and $U_2(k-1) \subset W(k-1)$ as in (a) and (b). Since $U_1(k-1)$ and $U_2(k-1)$ are disjoint, then there exists either an edge (i_k, j_1) with $j_1 \in U_1(k-1)$, $i_k \in V \setminus U_1(k-1)$, or an edge (i_k, j_2) with $j_2 \in U_2(k-1)$ and $i_k \in V \setminus U_2(k-1)$. Suppose that an edge (i_k, j_2) exists (the case of a edge (i_k, j_1) can be treated in a similar way). Only four cases are possible.

- (A) If $i_k \in W_1(k-1)$ and $W_1(k-1) \cup W_2(k-1) = V$, then we can terminate the algorithm and conclude that from any node $h \in U_1(k-1)$ there exists a path to all other nodes in the graph and thus there is a spanning tree.

(B) If $i_k \in W_1(k-1)$ and $W_1(k-1) \cup W_2(k-1) \neq V$, then set:

$$U_1(k) = U_1(k-1),$$

$$W_1(k) = W_1(k-1) \cup W_2(k-2),$$

$$U_2(k) = W_2(k) = \{h_k\},$$

where h_k is an arbitrary node not belonging to $W_1(k-1) \cup W_2(k-1)$.

(C) If $i_k \notin W_1(k-1) \cup W_2(k-1)$, then set

$$U_1(k) = U_1(k-1),$$

$$W_1(k) = W_1(k-1),$$

$$U_2(k) = \{i_k\},$$

$$W_2(k) = W_2(k-1) \cup \{i_k\}.$$

(D) If $i_k \in W_2(k-1) \setminus U_2(k-1)$ then

$$U_1(k) = U_1(k-1),$$

$$W_1(k) = W_1(k-1),$$

$$U_2(k) = U_2(k-1) \cup \{i_k\},$$

$$W_2(k) = W_2(k-1).$$

The algorithm terminates in a finite number of induction steps because at each step, except when finally case (A) holds true, either the number of nodes in $W_1 \cup W_2$ increases, or the number of nodes in $W_1 \cup W_2$ remains constant and the number of nodes in $U_1 \cup U_2$ increases. $\square$

1.8.3 Proofs of Propositions 1.33 and 1.35

Proof of Proposition 1.33. (ii) $\Rightarrow$ (i) We aim to show that there exist directed paths from any node to any other node. Fix $i \in \{1, \dots, n\}$ and let $R_i \subset \{1, \dots, n\}$ be the set of nodes that belong to directed paths originating from node i . Denote the unreachable nodes by $U_i = \{1,$

$\dots, n\} \setminus R_i$. We argue that U_i cannot contain any element, because if it does, then $R_i \cup U_i$ is a nontrivial partition of the index set $\{1, \dots, n\}$ and irreducibility implies the existence of a non-zero entry a_{jk} with $j \in R_i$ and $k \in U_i$. Therefore, $U_i = \emptyset$, and all nodes are reachable from i . The converse statement (i) = (ii) is proved similarly.

(i) $\Rightarrow$ (iii) If tt is strongly connected, then there exists a directed path of length $k \leq n - 1$ connecting any node i to any other node j . Hence, by Lemma 1.32(ii), the entry $(A^k)_{ij}$ is strictly positive. This immediately implies the statement (iii). The converse statement (iii) = (i) is proved similarly. $\square$

Next, we present a useful number theory result. This states that relatively co-prime numbers generate all sufficiently large natural numbers.

Lemma 1.83 (Natural number combination). *Let $a_1, \dots, a_N \in \mathbb{N}$ have greatest common divisor 1. There exists $k \in \mathbb{N}$ such that every number $m > k$ can be written as*

$$m = \alpha_1 a_1 + \dots + \alpha_N a_N,$$

for appropriate numbers $\alpha_1, \dots, \alpha_N \in \mathbb{N}$.

Proof. Assume that $a_1 \leq \dots \leq a_N$ without loss of generality. From the generalized Bezout identity we know that, for any numbers $a_1, \dots, a_N$ with greatest common divisor 1, there exist integers $\gamma_1, \dots, \gamma_N \in \mathbb{Z}$ such that

$$1 = \gamma_1 a_1 + \dots + \gamma_N a_N. \quad (1.8.1)$$

Pick $k = |\gamma_1| a_1^2 + \dots + |\gamma_N| a_N^2 \in \mathbb{N}$. Every number $m > k$ can be written as

$$m = k + m_{\text{qnt}} a_1 + m_{\text{rmndr}},$$

for appropriate numbers $m_{\text{qnt}} \geq 0$ and $0 \leq m_{\text{rmndr}} < a_1$. Using the definition of k and equation (1.8.1), we write

$$\begin{aligned} m &= |\gamma_1| a_1 + \dots + |\gamma_N| a_N + m_{\text{qnt}} a_1 + m_{\text{rmndr}} (\gamma_1 a_1 + \dots + \gamma_N a_N) \\ &= m_{\text{qnt}} a_1 + (|\gamma_1| a_1 + m_{\text{rmndr}} \gamma_1) a_1 + \dots + (|\gamma_N| a_N + m_{\text{rmndr}} \gamma_N) a_N. \end{aligned}$$

The proof is now completed by noting that each integer number $(|\gamma_1| a_1 + m_{\text{rmndr}} \gamma_1), \dots, (|\gamma_N| a_N + m_{\text{rmndr}} \gamma_N)$ is strictly positive, because $m_{\text{rmndr}} < a_1 \leq \dots \leq a_N$.

□

Proof of Proposition 1.35. (i) = (ii) Pick any i . We claim that there exists a number $k(i)$ with the property that, for all $m > k(i)$, we have that $(A^m)_{ii}$ is positive, that is, there exists a directed path from i to i of length m for all m larger than a number $k(i)$. To show this claim, let $\{c_1, \dots, c_N\}$

be the set of the cycles of tt and let $\{\ell_1, \dots, \ell_N\}$ be their

lengths. Because tt is aperiodic, Lemma 1.83 implies the existence of a number $h(\ell_1, \dots, \ell_N)$ such that any number larger than $h(\ell_1, \dots, \ell_N)$ is a linear combination of $\ell_1, \dots, \ell_N$ with natural numbers as coefficients. Because tt is strongly connected, there exists a path γ of arbitrary length $\Gamma(i)$ that starts at i , contains a vertex of each of the cycles $c_1, \dots, c_N$, and terminates at i . Now, we claim that $k(i) = \Gamma(i) + h(\ell_1, \dots, \ell_N)$ has the desired property. Indeed, pick any number $m > k(i)$ and write it as $k = \Gamma(i) + \beta_1 \ell_1 + \dots + \beta_N \ell_N$ for appropriate numbers $\beta_1, \dots, \beta_N \in \mathbf{N}$. A directed path from i to i of length m is constructed by attaching to the path γ the following cycles: β_1 times the cycle c_1 , β_2 times the cycle c_2 , $\dots$, β_N times the cycle c_N . Finally, having proved the existence of $k(i)$ with the desired property, let K be the maximum $k(i)$

over all nodes i , and recall that $\text{diam}(tt)$ is the maximum pairwise distance between nodes. Clearly, A^M is positive for all $M > K + \text{diam}(tt)$.

(ii) $\Rightarrow$ (i) From Lemma 1.32 we know that $A^k > 0$ means that there are paths from every node to every other node of length k . Hence, the digraph tt is strongly connected. Next, we prove aperiodicity. Because tt is strongly connected, each node of tt has at least one outgoing edge, that is, for all i , there exists at least one index j such that $a_{ij} > 0$. This fact implies that the matrix $A^{k+1} = AA^k$ is positive via the following simple calculation:
 $(A^{k+1})_{ij} = \sum_{h=1}^n a_{ih}(A^k)_{hj} \geq a_{ij}(A^k)_{jj} > 0$. In summary, we have shown

that, if A^k is positive for some k , then A^m is positive for all subsequent $m \geq k$. Therefore, there are cycles in tt of any length greater than or equal to k , which means that tt is aperiodic.

□

1.8.4 Proof of Theorem 1.37

Proof. We begin with statement (i). Let l_{ij} , for $i, j \in \{1, \dots, n\}$, be the entries of $L(tt)$. Note that $l_{ii} = \sum_{j=1}^n a_{ij} > 0$ and $l_{ij} \geq 0$ for $i \neq j$. By the Geršgorin disks Theorem 1.2, we know that each eigenvalue

of $L(tt)$ belongs to at least one of the disks

$$\left\{ z \in \mathbb{C} : |z - l_{ii}| \leq \sum_{j=1, j \neq i}^n |l_{ij}| \right\} = \left\{ z \in \mathbb{C} : |z - l_{ii}| \leq l_{ii} \right\}.$$

These disks contain the origin 0_n and complex numbers with a positive real part. This concludes the proof of statement (i).

Regarding statement (ii), note that $D_{\text{out}}(tt)$ is invertible because tt is strongly connected. Define the

two matrices $\tilde{A} = D_{\text{out}}(tt)^{-1}A(tt)$ and $\tilde{L} = D_{\text{out}}(tt)^{-1}L(tt)$, and note that they satisfy $\tilde{L} = I_n - \tilde{A}$. Since $D_{\text{out}}(tt)$ is diagonal, the matrices $A(tt)$ and $\tilde{A}$ have the same pattern of zeros and positive entries. This observation and the assumption that tt is strongly connected imply that A is nonnegative and irreducible. By the Perron–

Frobenius Theorem 1.11, the spectral radius $\rho(\tilde{A})$ is a simple eigenvalue.

Furthermore, one can verify that $\tilde{A}$ is row-stochastic (see Lemma 1.31),

and therefore, its spectral radius is 1 (see Exercise E1.4). In summary, we conclude that 1 is a simple eigenvalue of $\tilde{A}$, that 0 is a simple eigenvalue of $\tilde{L}$, that $\tilde{L}$ has rank $n - 1$, and that $L(tt)$ has rank $n - 1$.

Regarding statement (iii), we first prove that $\text{rank}(L(tt)) = n - 1$ implies the existence of a globally reachable vertex. By contradiction, let tt contain no globally reachable vertex. Then, by Lemma 1.27, there exist two nonempty disjoint subsets $U_1, U_2 \subset V(tt)$ without any out-neighbor. After

a permutation of the vertices, the adjacency matrix can be partitioned into the blocks

$$A(tt) = \begin{bmatrix} A_{11} & A_{12} & 0 \\ 0 & A_{21} & 0 \\ A_{31} & A_{32} & A_{33} \end{bmatrix}.$$

Here, A_{12} and A_{13} vanish because U_1 does not have any out-neighbor, and A_{21} and A_{23} vanish because U_2 does not have any out-neighbor. Note that $D_{11} - A_{11}$ and $D_{22} - A_{22}$ are the Laplacian matrices of the graphs defined by restricting tt to the vertices in U_1 and in U_2 , respectively. Therefore, the eigenvalue 0 has geometric multiplicity at least 2 for the matrix $D_{\text{out}}(tt) - A(tt)$. This contradicts the assumption that $\text{rank}(L(tt)) = n - 1$.

Next, still regarding statement (iii), we prove that the existence of a globally reachable vertex implies $\text{rank}(L(tt)) = n - 1$. Without loss of generality, we assume that tt contains self-loops at each node (so that D_{out} is invertible). Let R be the set of globally reachable vertices; let $r = 1, \dots, n$ be its cardinality. If $r = n$, then the graph is strongly connected and statement (ii)

implies that $\text{rank}(L(tt)) = n - 1$. Therefore, assume that $r < n$. Renumber the vertices so that R is the set of the first r vertices. After this permutation, the adjacency matrix and the Laplacian matrix can be partitioned into the blocks

$$A(tt) = \begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix}, \quad \text{and} \quad L(tt) = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}.$$

Here, $A_{12} \in \mathbb{R}^{r \times (n-r)}$ vanishes, because there can be no out-neighbor of R ; otherwise that out-neighbor would be a globally reachable vertex in $V \setminus R$. Note that the rank of $L_{11} \in \mathbb{R}^{r \times r}$ is exactly $r - 1$, since the digraph associated to A_{11} is strongly connected. To complete the proof it

suffices to show that the rank of $L_{22} \mathbf{R}^{(n-r) \times (n-r)}$ is full.

Note that the same block partition applies to the matrices $\tilde{A} = D^{-1} A$ and $\tilde{L} = D^{-1} L$ considered in the proof

of statement (ii) above. With this block decomposition, we compute

$$\tilde{A}^{n-1} = \begin{pmatrix} \Sigma & \tilde{A}_{n-1} & 0 \\ \tilde{A}_{11} & \tilde{A}_{21} & \tilde{A}_{22} \end{pmatrix},$$

for some matrix $\tilde{A}_{21}(n-1)$ that depends upon $\tilde{A}_{11}$, $\tilde{A}_{21}$ and $\tilde{A}_{22}$. Because a globally reachable node in tt is globally reachable also in the digraph associated to A , Proposition 1.33(v) implies that $\tilde{A}_{21}(n-1)$ is positive.

This fact, combined with the fact that $\tilde{A}$ and hence $\tilde{A}^{n-1}$ are row-stochastic, implies that $\tilde{A}^{n-1}$ has maximal row sum (that is, ∞ -induced norm) strictly less than 1. Hence, the spectral radii of $\tilde{A}^{n-1}$ and of $\tilde{A}_{22}$ are strictly less than 1. Since $\tilde{A}_{22}$ has spectral radius strictly less than 1, the matrix $\tilde{L}_{22} = I_{n-r} - \tilde{A}_{22}$, and in turn the matrix L_{22} , have full rank.

Regarding statement (iv), the equivalence between

(iv)a and (iv)b is 75

proved as follows. Because $\sum_{j=1}^n l_{ij} = d_{\text{out}}(v_i) - d_{\text{in}}(v_i)$ for all $i \in \{1, \dots, n\}$, it follows that $1_n^T L(tt) = 0^T$ if and only if $L_{\text{out}}(tt) = L_{\text{in}}(tt)$. prove that (iv)b implies (iv)c. Suppose that $L(tt)^T 1_n = 0^T$ and consider the system $\dot{\gamma}(t) = -L(tt)\gamma(t)$, $\gamma(0) = x_0$, together with the positive definite function $V : \mathbb{R}^n \rightarrow \mathbb{R}$ defined by $V(x) = x^T x$. We compute the Lie derivative of the function V along the vector field $x \mapsto -L(tt)x$ as $\dot{V}(x) = -2x^T L(tt)x$. Note that $\dot{V}(x) \leq 0$, for all $x \in \mathbb{R}^n$, is equivalent to $L(tt) + L(tt)^T \leq 0$. Because $1_n^T L(tt) = 0^T$ and $L(tt)1_n = 0_n$, it can immediately be established

that $\exp(-L(tt)t)$, $t \in \mathbb{R}$, is a doubly stochastic matrix. From Theorem 1.1, we know that if we let $\{P_a\}$ be the set of $n \times n$ permutation matrices, then there exist time-dependent convex combination coefficients $\lambda_a(t) = 1$, $\lambda_a(t) \geq 0$, such that $\exp(-L(tt)t) = \sum_a \lambda_a(t) P_a$. By the convexity of V and its invariance under coordinate permutations, for any $x \in \mathbb{R}^n$, we have

$$\begin{aligned} V(\exp(-L(tt)t)x) &= V\left(\sum_a \lambda_a(t) P_a x\right) \\ &\leq \sum_a \lambda_a(t) V(P_a x) = \sum_a \lambda_a(t) V(x) = V(x). \end{aligned}$$

In other words, $V(\exp(-L(tt)t)x) \leq V(x)$ for all $x \in \mathbb{R}^n$, which implies $\dot{V}(x) \leq 0$, for all $x \in \mathbb{R}^n$. Finally, we prove that (iv)c implies (iv)b. By assumption, $-x^T (L(tt) + L(tt)^T)x = -2x^T L(tt)x \leq 0$ for all $x \in \mathbb{R}^n$. In particular, for any small $\varepsilon > 0$ and $x = 1_n - \varepsilon L(tt)^T 1_n$,

$$-(1_n - \varepsilon 1_n^T L(tt))L(tt)(1_n - \varepsilon L(tt)^T 1_n) = \varepsilon^2 1_n^T L(tt)^2 1_n + O(\varepsilon) \leq 0,$$

which is possible only if $L(tt)^T 1_n = 0^T$. □

1.8.5 Proofs of Theorem 1.63 and Proposition 1.67

In this section, we prove Theorem 1.63. The exposition follows along the main lines of the original proof by Moreau (2005), with the variation of using the LaSalle

Invariance Principle for set-valued dynamical systems, presented as Theorem 1.21. We begin with some preliminary results.

Lemma 1.84 (Union of digraphs and sums of adjacency matrices). *Let $tt_1, \dots, tt_\delta$ be unweighted digraphs with common node set $\{1, \dots, n\}$ and adjacency matrices $A_1, \dots, A_\delta$. The unweighted digraph*

$$tt_1 \cup \dots \cup tt_\delta = (\{1, \dots, n\}, E(A_1) \cup \dots \cup E(A_\delta))$$

is equal to the unweighted digraph associated to the nonnegative matrix

$$\sum_{k \in \{1, \dots, \delta\}} A_k, \text{ that is, the unweighted digraph } (\{1, \dots, n\}, E(A_1 + \dots + A_\delta)).$$

Proof. If $(i, j) \in \cup_{k \in \{1, \dots, \delta\}} E(tt_k)$, then there exists $k_0 \in \{1, \dots, \delta\}$ such

that $(i, j) \in E(tt_{k_0})$. Denoting the entries of the matrix A_k by $a_{ij}(k)$, this implies that $a_{ij}(k_0) > 0$, that $a_{ij}(1) + \dots + a_{ij}(\delta) > 0$, and that (i, j) is an edge in $E(A_1 + \dots + A_\delta)$. The converse statement is easily proved with an analogous reasoning.

□

In what follows, for $\alpha \in]0, 1]$, let $F(\alpha)$ denote the set of $n \times n$ stochastic matrices that are non-degenerate with respect to α . Given $\alpha \in]0, 1]$ and $\delta \in \mathbb{N}$, define the sets $F_\delta(\alpha) \subset \mathbb{R}^{n \times n}$ by

$$F_\delta(\alpha) = \{F \in F(\alpha) \mid \exists F_1, \dots, F_\delta \in F(\alpha) \text{ such that } F = F_\delta \cdot \dots \cdot F_1$$

and $tt(F_1) \cup \dots \cup tt(F_\delta)$ contains a globally reachable node ,

or, equivalently by Proposition 1.33,

$$F_\delta(\alpha) = \{F \in F(\alpha) \mid \exists F_1, \dots, F_\delta \in F(\alpha) \text{ such that } F = \sum_{n=1}^n F_\delta \dots F_1$$

and a column of $(F_1 + \dots + F_\delta)$ has positive entries $\sum$.

Lemma 1.85 (Compact sets of stochastic matrices). *For $\alpha \in]0, 1]$, the sets $F(\alpha)$ and $F_\delta(\alpha)$, $\delta \in \mathbb{N}$, are compact.*

Proof. All sets are clearly bounded. In Exercise E1.24, we invite the reader

to prove that $F(\alpha)$ is closed. Let us now prove that $F_\delta(\alpha)$ is closed. Consider a matrix sequence $\{F(k) \mid k \in \mathbb{N}\} \subset F_\delta(\alpha)$ convergent to some matrix F . Because $F(\alpha^\delta)$ is closed, we establish that $F \in F(\alpha^\delta)$. Because each matrix $F(k)$ belongs to $F_\delta(\alpha)$, there exist matrices $F_1(k), \dots, F_\delta(k) \in F(\alpha)$ such that $F(k) = F_\delta(k) \cdot \dots \cdot F_1(k)$. We claim that there exists a sequence $k_l \in \mathbb{N}$, for $l \in \mathbb{N}$, such that, for all $s \in \{1, \dots, \delta\}$, the matrix sequences $F_s(k_l)$, $l \in \mathbb{N}$, are convergent. (To see this, note that $F_1(k)$ takes value in a compact set;

hence it must have a convergent subsequence. Restrict $F_2(k)$ to the instants of time in the convergent

subsequence for $F_1(k)$ and observe that it takes value in a compact set, etc.) Therefore, there exist matrices F_s , to which the matrix sequences $F_s(k_l)$, $l \in \mathbf{N}$, converge. Taking the limit as $l \rightarrow +\infty$ in the equality $F(k_l) = F_s(k_l) \cdot \dots \cdot F_1(k_l)$, we establish that $F = F_s \cdot \dots \cdot F_1$. Finally, it remains to be shown that a column of $B := (F_1 + \dots + F_s)^n$ has positive entries. For $k \in \mathbf{N}$, define $B(k) = (F_1(k) + \dots + F_s(k))^n$. Clearly, $B(k) \rightarrow B$ as $k \rightarrow +\infty$. By the definition of the sequence $F(k)$, each $B(k) = (b_{ij}(k))$ has the property that there exists $j_k \in \{1, \dots, n\}$ such that $b_{ij_k}(k) > 0$ for all $i \in \{1, \dots, n\}$. Since $\{1, \dots, n\}$ is a finite set, there exists $j_0 \in \{1, \dots, n\}$ that satisfies this property for an infinite subsequence of matrices $B(k_l)$, $l \in \mathbf{N}$. With some straightforward bookkeeping, we write

$$(B(k_l))_{ij_0} = \sum_{a_1, \dots, a_n=1}^{\delta} \sum_{\substack{h_1=1 \\ \vdots \\ h_{n-1}=1}}^n \sum_{h_{n-1}=1}^n (F_{a_1}(k_l))_{ih_1} \dots (F_{a_n}(k_l))_{h_{n-1}j_0}.$$

Note that, because $F_s(k) \in F(\alpha)$, for $s \in \{1, \dots, \delta\}$, each nonzero entry $F_s(k)$ is lower bounded by $\alpha > 0$. Furthermore, each entry $(B(k_l))_{ij_0}$ is the sum of nonnegative terms, each of which is the product of n factors, each of which is lower bounded by α . Hence, because $(B(k_l))_{ij_0}$ is positive, it is also lower bounded by α^n . Since $\lim_{l \rightarrow +\infty} B(k_l) = B$, by the compactness of $[\alpha^n, 1] \cup \{0\}$, it must be that $B = (b_{ij})$ satisfies $b_{ji_0} \geq \alpha$ for all $j = 1, \dots, n$. In particular, this implies that $F_{\delta}(\alpha)$ and then $F_{\delta}(\alpha)$ is closed. $\square$

Finally, we are able to prove the equivalences in Theorem 1.63.

Proof of Theorem 1.63. First, we prove that (i) implies (ii). Suppose that for all durations $\delta \in \mathbb{N}$, there exists some $\ell_0 \in \mathbb{N}$ such that the digraph with edges $\bigcup_{s \in [\ell_0, \ell_0 + \delta]} E(F(s))$ does not contain a globally reachable node. By Lemma 1.27, there must exist a set of nodes $U_1, U_2 \subset \{1, \dots, n\}$ such that there are no out-going edges (i_1, j_1) , with $i_1 \in U_1, i_1 \in \{1, \dots, n\} \setminus U_1$ or (i_2, j_2) , with $j_2 \in U_2, i_2 \in \{1, \dots, n\} \setminus U_2$. Take any values $a, b \in \mathbb{R}, a = b$, and consider the initial conditions:

$$w_i(\ell_0) = \begin{cases} a, & i \in U_1, \\ b, & i \in U_2, \\ c \in \text{co}(a, b), & i \in \{1, \dots, n\} \setminus (U_1 \cup U_2). \end{cases}$$

Because of the properties of U_1 and U_2 , for all $\delta \in \mathbb{N}$, we must have

$$w_j(\ell_0 + \delta + 1) = \begin{cases} a, & j \in U_1, \\ b, & j \in U_2, \end{cases}$$

$$\square \quad c \in \text{co}(a, b), \quad j \in \{1, \dots, n\} \setminus (U_1 \cup U_2).$$

Because δ can be chosen arbitrarily large, one can easily construct a contradiction with the fact that $\text{diag}(\mathbf{R}^n)$ is supposed to be uniformly globally attractive.

Next, we show that (ii) implies (i). Let $\alpha \in]0, 1]$ to be the scalar with respect to which the sequence is non-degenerate. Consider the set-valued discrete-time dynamical system $(\mathbf{R}^n, \mathbf{R}^n, T_{\alpha, \delta})$, with evolution map $T_{\alpha, \delta}$

$\mathbf{R}^n \Rightarrow \mathbf{R}^n$ defined by

$$T_{\alpha, \delta}(w) = \{Fw \mid F \in \mathbf{F}_\delta(\alpha)\}.$$

Because of this definition, any trajectory $w : \mathbb{Z}_{\geq 0} \rightarrow \mathbf{R}^n$ of the averaging algorithm (1.6.2) satisfies

$$w((k+1)\delta) \in T_{\alpha, \delta}(w(k\delta)), \quad k \in \mathbb{Z}_{\geq 0}.$$

In what follows, we intend to use the LaSalle Invariance Principle for set-valued discrete dynamical systems, presented as Theorem 1.21, to prove that $\lim_{l \rightarrow +\infty} \text{dist}(w(kl), \text{diag}(R^n)) = 0$. This will then imply, by Lemma 1.24, the

uniform attractivity statement in the theorem. In the following, we check the conditions of the theorem.

Closedness of the set-valued dynamical system. Consider a pair of vector sequences $\{x_k \mid k \in \mathbb{N}\}$ and $\{y_k \mid k \in \mathbb{N}\}$ in $\mathbb{R}^n$ such that $\lim_{k \rightarrow +\infty} x_k =$

x , $\lim_{k \rightarrow +\infty} y_k = y$, and $y_k \in T_{a, \delta}(x_k)$, for all $k \in \mathbb{N}$. We need to show

that $y \in T_{a, \delta}(x)$. By definition of $T_{a, \delta}$ and because $y_k \in T_{a, \delta}(x_k)$, there exists a sequence $\{F(k) \mid k \in \mathbb{N}\} \subseteq F_\delta(\alpha)$ such that $F(k)x_k = y_k$ for all $k \in \mathbb{N}$. Furthermore, since $F_\delta(\alpha)$ is compact by Lemma 1.85, there exists a subsequence $F(k_l) \mid \mathbb{N}$ that is convergent to some $F_\delta(\alpha)$. The desired conclusion follows from

$$y = \lim_{l \rightarrow +\infty} y_{k_l} = \lim_{l \rightarrow +\infty} F(k_l)x_{k_l} = Fx.$$

Non-increasing Lyapunov function. Define the function $V : \mathbb{R}^n \rightarrow \mathbb{R}_{\geq 0}$ by

$$V(x) = \max_{i \in \{1, \dots, n\}} x_i - \min_{i \in \{1, \dots, n\}} x_i.$$

Note that V is continuous. Pick any $x \in \mathbb{R}^n$ and any stochastic matrix $F \in F_\delta(\alpha)$. Recall that $\|x\|_\infty = \max_{i \in \{1, \dots, n\}} |x_i|$, and that $\|F\|_\infty = 1$. Therefore, by the definition of the induced norm, $\|Fx\|_\infty \leq \|x\|_\infty$. Similarly, in components,

$$(Fx)_i = \sum_{j \in \{1, \dots, n\}} f_{ij}x_j \geq \sum_{j \in \{1, \dots, n\}} \min_{k \in \{1, \dots, n\}} f_{ij}x_k,$$

which implies $\min_{i \in \{1, \dots, n\}} (F x)_i \geq \min_{k \in \{1, \dots, n\}} x_k$. Therefore, we have that $V(Fx) \leq V(x)$ for all $x \in \mathbb{R}^n$ and $F \in \mathcal{F}_\delta(\alpha)$. In other words, the function V is non-increasing along $T_{\alpha, \delta}$ in $\mathbb{R}^n$.

Boundedness. It can immediately be seen that, since $Fx \leq x$ for all stochastic matrices F and vectors x , the trajectory $k \mapsto w(k\delta)$ is bounded.

Invariant set. By Theorem 1.21, any trajectory of $T_{\alpha, \delta}$, and hence also the trajectory $w : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ of the averaging algorithm (1.6.2), will converge to the largest weakly positively invariant set contained in a level set of the Lyapunov function V and in a set where the Lyapunov function does not decrease along T . In the following, we determine that this set must be contained in $\text{diag}(\mathbb{R}^n)$.

For $k \in \mathbb{N}$ fixed, assume that $w(k\delta)$ satisfies $V(w(k\delta))$

> 0 . Given the av- 79

eraging algorithm (1.6.2) defined by the sequence $\{F(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\} \subset F(\alpha)$, define $F_1(k) = F(k+1), \dots, F_\delta(k) = F(k+\delta)$. Additionally, define $F(k) = F_\delta(k) \cdot \dots \cdot F_1(k)$ and note that $F(k) \in F(\alpha)$, by construction. With this notation, note that $w(k\delta+s) = F_s(k) \cdot \dots \cdot F_1(k)w(k\delta)$ for $s \in \{1, \dots, \delta\}$. Define $w_M = \max_{i \in \{1, \dots, n\}} w_i(k\delta)$ and $w_m = \min_{i \in \{1, \dots, n\}} w_i(k\delta)$; by hypothesis we know $w_M > w_m$. Define $U_M = \{i \in \{1, \dots, n\} \mid w(k\delta) = w_M\}$ and $U_m = \{i \in \{1, \dots, n\} \mid w(k\delta) = w_m\}$; by hypothesis we know $U_M \cap U_m = \emptyset$.

Now, we are ready to use property (ii) in the theorem statement. Since $(\{1, \dots, n\}, \cup_{s \in \{1, \dots, \delta\}} E(F_s(k)))$ contains a globally reachable node and since U_M and U_m are nonempty and disjoint, then Lemma 1.27 implies that there exists either

- (an out-neighbor of U_M) an edge $(i_M, j_M) \in E(F_s(k\delta))$ with $i_M \in U_M$, $j_M \in \{1, \dots, n\} \setminus U_M$, and $s \in \{1, \dots, \delta\}$; or
- (an out-neighbor of U_m) an edge $(i_m, j_m) \in E(F_s(k\delta))$ with $i_m \in U_m$, $j_m \in \{1, \dots, n\} \setminus U_m$, and $s \in \{1, \dots, \delta\}$.

Without loss of generality, suppose that an edge (i_M, j_M) exists and let $s_0 \in \{1, \dots, \delta\}$ be the first time index for which this happens. We have the following two facts.

First, for every $s \in \{1, \dots, s_0 - 1\}$, there does not exist any edge (i, h) with $i \in U_M$ and $h \notin U_M$, and, thus, for all $i \in U_M$;

$$\begin{aligned} w_i(k\delta + 1) &= \sum_{j=1}^n (F_1(k))_{ij} w_j(k\delta) = \sum_{h \in U_M} (F_1(k))_{ih} w_h(k\delta) \\ &= \sum_{h \in U_M} (F_1(k))_{ih} w_M = w_M. \end{aligned}$$

The same argument can be repeated for $F_2(k), \dots, F_s(k)$, so that $w_i(k\delta + s) = w_M$ for all $i \in U_M$.

Second, if $i \notin U_M$ at time $k\delta$, then $w_i(k\delta+s) < w_M$ for all $s \in \{1, \dots, s_0 - 1\}$. To see this, we compute

$$\begin{aligned}
w_i(k\delta + 1) &= \sum_{j=1}^n (F_1(k))_{ij} w_j(k\delta) = (F_1(k))_{ii} w_i(k\delta) + \sum_{j=1, j \neq i}^n (F_1(k))_{ij} w_j(k\delta) \\
&\leq (F_1(k))_{ii} w_i(k\delta) + \sum_{j=1, j \neq i}^n (F_1(k))_{ij} w_M \\
&\leq \alpha w_i(k\delta) + (1 - \alpha) w_M < w_M,
\end{aligned}$$

where we used the assumption of non-degeneracy with parameter $\alpha \in]0, 1]$.

The same argument can be repeated for the subsequent multiplications by the matrices $F_2(k), \dots, F_s(k)$.

We finally reach time s_0 and compute

$$\begin{aligned}
 w_{i_M}(k\delta + s_0) &= \sum_{j=1}^n (F_{s_0}(k))_{i_M j} w_j(k\delta + s_0 - 1) \\
 &= (F_{s_0}(k))_{i_M j_M} w_{j_M}(k\delta + s_0 - 1) + \sum_{j=1, j \neq j_M}^n (F_{s_0}(k))_{i_M j} w_j(k\delta + s_0 - 1) \\
 &< (F_{s_0}(k))_{i_M j_M} w_M + \sum_{j=1, j \neq j_M}^n (F_{s_0}(k))_{i_M j} w_j(k\delta + s_0 - 1) \leq w_M.
 \end{aligned}$$

This implies that $w_{i_M}((k+1)\delta) < w_M$, so that i_M does not belong to U_M at time $(k+1)\delta$. That is, the cardinality of U_M decreases at least by one after $(k+1)\delta$. Since $\{1, \dots, n\}$ is finite, after repeating this argument at most $n-1$ times, we have that U_M becomes empty at time $(k+n-1)\delta$. (Here we are assuming that the out-neighbor always exists for U_M ; an analogous argument can be made for the general case.) This is enough to guarantee that $V(w((k+n)\delta)) < w_M - w_m = V(w(k\delta))$. This is what we need to conclude that $\lim_{k \rightarrow +\infty} \text{dist}(w(k\delta), \text{diag}(R)) = 0$. In summary, this concludes the proof of Theorem 1.63. $\square$

We conclude this section by establishing convergence to an individual point, rather than a set of points.

Proof of Proposition 1.67. We adopt the same notation as above, that is, as in the proof of Theorem 1.63. Since $F(k)_{i_M j_M} > 0$, the set of sequence points $w(k\delta) \mid k \in \mathbb{N}$ belongs to the convex hull of all the components of the initial

condition, that is, $[\min_i w_i(o), \max_i w_i(o)]^n$. Since $[\min_i w_i(o), \max_i w_i(o)]^n$ is compact, there exists a convergent subsequence $\{w(k_l\delta) \mid l \in \mathbb{N}\}$ to a point $c1_n$. We also notice that for any $k_l \in \mathbb{N}$, we have $w_i((k_l + k)\delta) \in [\min_i w_i(k_l\delta), \max_i w_i(k_l\delta)]^n$, for all $i \in \{1, \dots, n\}$ and $k \in \mathbb{N}$. Because $\lim_{l \rightarrow +\infty} w(k_l\delta) = c1_n$ we know that $\lim_{l \rightarrow +\infty} [\min_i w_i(k_l\delta), \max_i w_i(k_l\delta)]^n = c1_n$. Therefore, any sequence $\{w((k_l + k)\delta) \mid k \in \mathbb{N}\}$, for $l \in \mathbb{N}$, must converge to $c1_n$. This implies that $\lim_{k \rightarrow +\infty} w(k\delta) = c1_n$. $\square$

1.8.6 Proofs of Theorems 1.79 and 1.80

Proof of Theorem 1.79. Let us prove fact (i). Because $\text{Trid}_n(a, b, a)$ is a real symmetric matrix, $\text{Trid}_n(a, b, a)$ is normal and its 2-induced norm—that is, its largest singular value—is equal to the magnitude of its eigenvalue with

the largest magnitude. Based on this information and on the eigenvalue computation in Lemma 1.77, we compute

$$\begin{aligned} \|\text{Trid}_n(a, b, a)\|_2 &= \max_{i \in \{1, \dots, n\}} |b + 2a \cos \frac{i\pi}{n+1}| \\ &\leq |b| + 2|a| \max_{i \in \{1, \dots, n\}} \left| \cos \frac{i\pi}{n+1} \right| \leq |b| + 2|a| \cos \frac{\pi}{n+1}. \end{aligned}$$

Because we assumed that $|b| + 2|a| = 1$ and $\cos \frac{\pi}{n+1} < 1$ for any $n \geq$

2, the 2-induced norm of $\text{Trid}_n(a, b, a)$ is strictly less than 1.

Additionally, for $\ell > 0$, we bound from above the magnitude of the curve x , as

$$\|x(\ell)\|_2 = \|\text{Trid}_n(a, b, a) x\|_2 \leq \left(|b| + 2|a| \cos \frac{\pi}{n+1} \right)^{\sum_{l=0}^{\ell-1} 1} \|x\|_2.$$

In order to have $\|x(\ell)\|_2 < \varepsilon \|x_0\|_2$, it is sufficient to require that $\log \varepsilon > \ell \log \left(|b| + 2|a| \cos \frac{\pi}{n+1} \right)$, that is,

$$\ell > \frac{\log \varepsilon^{-1}}{-\log \left(|b| + 2|a| \cos \frac{\pi}{n+1} \right)}. \quad (1.8.2)$$

The upper bound now follows by noting that, as $t \rightarrow 0$, we have

$$-\log(1 - 2|a|(1 - \cos t)) = |a|t^2 + O(1).$$

Let us now show the lower bound. Assume, without loss of generality, that

$ab > 0$ and consider the eigenvalue $b + 2a \cos(\frac{\pi}{n+1})$ of $\text{Trid}_n(a, b, a)$. Note

that $|b + 2a \cos(\frac{\pi}{n+1})| = |b| + 2|a| \cos(\frac{\pi}{n+1})$. (If $ab \leq 0$, then consider the

eigenvalue $b + 2a \cos(\frac{n\pi}{n+1})$.) For $n > 2$, define the unit-length vector

$$v_n = \frac{1}{\sqrt{2}} \begin{bmatrix} \sin \frac{\pi}{n+1} \\ \vdots \\ \sin \frac{n\pi}{n+1} \end{bmatrix} \in \mathbb{R}^n, \quad (1.8.3)$$

and note that, by Lemma 1.77(i), v_n is an eigenvector of $\text{Trid}_n(a, b, a)$ with eigenvalue $b + 2a \cos(\frac{\pi}{n+1})$. The trajectory x with initial condition v_n sat-

isfies $\|x(\ell)\|_2 = \sqrt{|b| + 2|a| \cos \frac{\ell\pi}{n+1}} \|v_n\|_2$, and therefore, it will enter $B(\frac{1}{2}, \sqrt{2} v_n)$ only when ℓ satisfies equation (1.8.2). This completes the proof of fact (i).

Next, we prove fact (ii). Clearly, all eigenvalues of the matrix $\text{Trid}_n(a, b, 0)$ are strictly inside the unit disk. For $\ell > 0$, we compute

$$\begin{aligned} \text{Trid}_n(a, b, 0)^\ell &= b^\ell I_n + \sum_{j=0}^{\ell-1} \frac{a^j \ell!}{j!(\ell-j)!} \text{Trid}_n(1, 0, 0)^j, \end{aligned}$$

because of the nilpotency of $\text{Trid}_n(1, 0, 0)$. Now, we can bound from above the magnitude of the curve x , as

$$\begin{aligned} \|x(\ell)\|_2 &= \|\text{Trid}_n(a, b, 0)^\ell x_0\|_2 \\ &\leq |b|^\ell \sum_{j=0}^{\ell-1} \frac{a^j \ell!}{j!(\ell-j)!} \cdot \sum_{j=0}^{\ell-1} \|\text{Trid}_n(1, 0, 0)^j x_0\|_2 \leq e^{a/b} \ell^{n-1} |b|^\ell \|x_0\|_2. \end{aligned}$$

Here, we used $\|\text{Trid}_n(1, 0, 0)^j x_0\|_2 \leq \|x_0\|_2$ and $\max_{j \in \{0, \dots, n-1\}} \frac{\ell!}{j!(\ell-j)!} \leq \ell^{n-1}$. Therefore, in order to have $\|x(\ell)\|_2 < \varepsilon \|x_0\|_2$, it suffices that $\log(e^{a/b}) + (n-1) \log \ell + \ell \log |b| \leq \log \varepsilon$, that is,

$$\ell - \frac{n-1}{\log |b|} \log \ell > \frac{a}{b - \log \varepsilon}.$$

A sufficient condition for $\ell - \frac{n-1}{\log |b|} \log \ell > \beta$, for $a, \beta > 0$, is that $\ell \geq 2\beta + 2\alpha \max\{1, \log \alpha\}$. For, if $\ell \geq 2\alpha$, then $\log \ell$ is bounded from above by the line $\ell/2\alpha + \log \alpha$. Furthermore, the line $\ell/2\alpha + \log \alpha$ is a lower bound for the line $(\ell - \beta)/\alpha$ if $\ell \geq 2\beta + 2\alpha \log \alpha$. In summary, it is true that

$$\|x(\ell)\|_2 \leq \varepsilon \|x_0\|_2 \text{ whenever } \ell \geq 2 \frac{a - \log \varepsilon}{- \log |b|} + 2 \frac{n-1}{- \log |b|} \max\left\{1, \log \frac{n-1}{- \log |b|}\right\}.$$

This completes the proof of the upper bound, that is, fact (ii).

The proof of fact (iii) is similar to that of fact (i).

Because $\text{Circ}_n(a, b, c)$ is circulant, it is also normal and each of its singular values corresponds to an eigenvector–eigenvalue pair. From Lemma 1.77(ii) and from the assumption $a + b + c = 1$, it is clear that the eigenvalue corresponding to $i = n$ is equal to 1; this is the largest singular value of $\text{Circ}_n(a, b, c)$ and the corresponding eigenvector is 1_n . We now compute the second largest singular value:

$$\max_{i \in \{1, \dots, n-1\}} \left| b + (a + c) \cos \frac{i2\pi}{n} + \sqrt{-1}(c - a) \sin \frac{i2\pi}{n} \right|_{\mathbb{C}}$$

$$= \left| \frac{2\pi}{n} - (a + c) \left(1 - \cos \frac{2\pi}{n} \right) + \sqrt{-1}(c - a) \sin \frac{2\pi}{n} \right|_{\mathbb{C}}$$

Here $\|\cdot\|_{\mathbb{C}}$ is the norm in $\mathbb{C}$. Because of the assumptions on a, b, c , the second largest singular value is strictly less than 1. In the orthogonal decomposition

induced by the eigenvectors of $\text{Circ}_n(a, b, c)$, we assume that the vector y_0 has a component y_{ave} along the eigenvector 1_n . For $\ell > 0$, we bound the distance of the curve $y(\ell)$ from $y_{\text{ave}}1_n$ as

$$\begin{aligned} & \|y(\ell) - y_{\text{ave}}1_n\|_2 \\ &= \|\text{Circ}_n(a, b, c) y_0 - y_{\text{ave}}1_n\|_2 = \|\text{Circ}_n(a, b, c) \cdot y_0 - y_{\text{ave}}1_n\|_2 \\ &\leq \frac{1}{\sqrt{1 - (a+c) \cos \frac{2\pi}{n} + \sqrt{1 - (c-a) \sin \frac{2\pi}{n}}}} \|y_0 - y_{\text{ave}}1_n\|_2. \end{aligned}$$

This proves that $\lim_{\ell \rightarrow +\infty} y(\ell) = y_{\text{ave}}1_n$. Also, for $\alpha = a + c, \beta = c - a$ and as $t \rightarrow 0$, we have

$$\frac{1}{\log \left(1 - \alpha(1 - \cos t) + \beta^2 \sin^2 t \right)^{1/2}} = \frac{1}{(\alpha - \beta^2)t^2} + O(1).$$

Here, $\beta^2 < \alpha$ because $a, c \in]0, 1[$. From this, one deduces the upper bound in (iii).

Now, consider the eigenvalues $\lambda_n = b + (a+c) \cos \frac{2\pi}{n} + \sqrt{1 - (c-a) \sin \frac{2\pi}{n}}$ of $\text{Circ}_n(a, b, c)$, and its associated eigenvectors $v_n = \begin{pmatrix} 1 \\ \omega \\ \vdots \\ \omega^{n-1} \end{pmatrix} \in \mathbb{C}^n$, $\bar{v}_n = \begin{pmatrix} 1 \\ \omega^{n-1} \\ \vdots \\ \omega \end{pmatrix} \in \mathbb{C}^n$.

$$v_n + \bar{v}_n \text{ satisfies } \|y(\ell)\|_2 = \|\lambda_n v_n + \bar{\lambda}_n \bar{v}_n\|_2 = \sqrt{|\lambda_n|^2 + |\bar{\lambda}_n|^2}, \text{ and therefore}$$

Note that the vector $\frac{1}{\sqrt{2}}(v_n + \bar{v}_n)$ belongs to $\mathbb{R}^n$. Moreover, its component along the eigenvector 1_n is zero. The trajectory y with initial condition

$y_0 = \frac{1}{\sqrt{2}}(v_n + \bar{v}_n)$ satisfies $\|y(\ell)\|_2 = \frac{1}{\sqrt{2}} \sqrt{|\lambda_n|^2 + |\bar{\lambda}_n|^2}$, and therefore

it will enter $B(0_n, \varepsilon \|v_n + \bar{v}_n\|_2)$ only when

$$\ell > \frac{\log \varepsilon^{-1}}{-\log 1 - (a+c) \cos \frac{2\pi}{n} + \sqrt{1 - (c-a) \sin \frac{2\pi}{n}}}.$$

This completes the proof of fact (iii). □

Proof of Theorem 1.80. We prove fact (i) and observe that the proof of fact (ii) is analogous. Consider the change of coordinates

$$\begin{pmatrix} x(\ell) \\ y(\ell) \end{pmatrix} = P_+ \begin{pmatrix} x'_{\text{ave}}(\ell) \\ y(\ell) \end{pmatrix} = x'_{\text{ave}}(\ell) \mathbf{1}_n + P_+ \begin{pmatrix} 0 \\ y(\ell) \end{pmatrix}$$

where $x'_{\text{ave}}(\ell) \in \mathbb{R}$ and $y(\ell) \in \mathbb{R}^{n-1}$. A quick calculation shows that $x'_{\text{ave}}(\ell) = \frac{1}{n} \mathbf{1}_n^T x(\ell)$, and the similarity transformation described in equation (1.6.7)

implies

$$y(\ell+1) = \text{Trid}_{n-1}(a, b, a)y(\ell), \quad \text{and} \quad x'_{\text{ave}}(\ell+1) = (b+2a)x'_{\text{ave}}(\ell).$$

Therefore, $x_{\text{ave}} = x'_{\text{ave}}$. It is also clear that

$$\begin{aligned} x(\ell+1) - x_{\text{ave}}(\ell+1)1_n \\ = P_+ y(\ell+1) - P_+ x_{\text{ave}}(\ell+1)1_n \\ = P_+ (y(\ell+1) - x_{\text{ave}}(\ell+1)1_n) \\ = P_+ (P_+^{-1} (x(\ell) - x_{\text{ave}}(\ell)1_n)). \end{aligned}$$

0

$0 \text{ Trid}_{n-1}(a, b, a)$ $+$
 Consider the matrix in parentheses determining the trajectory $\ell \mapsto (x(\ell) - x_{\text{ave}}(\ell)1_n)$. This matrix is symmetric, its singular values are 0 and the singular values of $\text{Trid}_{n-1}(a, b, a)$, and its eigenvectors are 1_n and the eigenvectors of $\text{Trid}_{n-1}(a, b, a)$ (padded with an extra zero). These facts are sufficient to duplicate, step by step, the proof of fact (i) in Theorem 1.79. Therefore, the trajectory $\ell \mapsto (x(\ell) - x_{\text{ave}}(\ell)1_n)$ satisfies the stated properties.

4. Research Overview

Section III

Middleware Using Distributed Object-Oriented Components



Taylor & Francis

Taylor & Francis Group

<http://taylorandfrancis.com>

8 Distributed Object-Oriented Components

8.1 JUST WHAT DO WE MEAN BY “OBJECT-ORIENTED MIDDLEWARE” AND “COMPONENT MIDDLEWARE,” ANYWAY?

We previously defined Middleware back in Chapter 1. As you recall also from Chapter 1, the first attempts at middleware were things like sockets and Sun’s Remote Procedure Calls. Those were functionally oriented because they came along before the object-oriented paradigm was widely used in industry. Functionally oriented means they focused on procedure calls—data was separate. In the object-oriented paradigm, of course, data and procedure calls are encapsulated inside an object.

So as industry moved toward using the object-oriented paradigm in the late 1980s and early 1990s, a need to have distributed objects talk to one another arose. One of the best known early technologies was Common Object Request Broker Architecture (CORBA) which dates back to 1991. Surprisingly, CORBA is still very much with us, although in terms of new development it has become more of a niche product today, targeted toward embedded systems. We will look in depth at CORBA in Section 8.2.

As the 1990s progressed, other object-oriented middlewares arose. Another well-known middleware was, or still is (because to my understanding it is still supported) Microsoft’s Distributed Component Object Model (DCOM). Both CORBA and DCOM were originally expected to be major players in the internet. However, when firewalls became necessary, these technologies fell behind, largely because they didn’t work well through firewalls.

In the early 2000s, when Microsoft brought out their .NET framework, they also brought out .NET Remoting, which was intended to solve some of the problems originally experienced by DCOM. We will discuss .NET Remoting in depth in Section 8.3. More recently, however, although .NET Remoting is still supported in Windows, Microsoft has deprecated new development .NET Remoting in favor of Windows Communication Foundation (WCF). We will look at WCF in future chapters (Chapters 10 and 12).

Both CORBA and .NET Remoting are intended to handle client-side issues as well as server-side issues such as persistence, object lifetime, etc.

In the late 1990s, Enterprise Java Beans (EJBs) was developed, primarily by Sun Microsystems. EJBs are primarily targeted toward server-side issues such as persistence and object lifetime, although considerable client-side processing (such as asynchronous client support) is also included. We will look at EJBs in Section 8.4.

All three of these technologies can be considered “component-based” technologies, speaking in general terms. Generally speaking, component-based software engineering consists of combining loosely coupled components into systems. There have been arguments about whether there is, in fact, a difference between component-based systems and object-oriented systems. The general argument is that objects in an object-oriented system focus modeling real world situations, whereas instead components in a component-based system are focused solely on combining existing components into systems. (But where do the existing components come from in the first place goes the other argument.)

However, there is a bigger distinction between a component and an object, for EJBs. EJBs on the server side do not run independently, but rather exist inside a container inside an application server. These server-side EJBs would receive all

communication through the container. In some EJB

143

terminology, this makes a server-side EJB a component, while an object that does not run inside a container is not a component.

In the following sections, we will look at CORBA (languages used are C++ and Java), at .NET Remoting (in the C# language), and EJBs (in Java). So you'll see how to do middlewares for a wide swath of object-oriented languages. Note that I've tried to write each of the sections so you can read them independently. For the main purpose of this textbook, you will want to read all these sections so you can compare. But for programming purposes, you want to have what you need in the appropriate section. ☺

As you read these sections, think about the capabilities and limitations of each of these middle- wares versus the others.

For each of these technologies, there is an associated larger practical example provided in Chapter 17 (CORBA, .Net Remoting) or Chapter 18 (EJB). Chapters 17 and 18 are available in online resources.

8.2 TECHNOLOGY REVIEW: COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA)

Common Object Request Broker Architecture (CORBA) is a mature technology, having origin- ally been developed in the early 1990s: the first version of the CORBA specification came out in October, 1991. So at this writing, CORBA has been around for a quarter century! This timeframe doesn't compare to the C language or the TCP/IP protocols, which date from the early 1970s, but as most technologies go, this is a very respectable lifetime—and it's not over yet. ☺ CORBA is still a major force in the embedded systems world, considerable new development is being done there (Schuster

2012). Also, a large base of installed CORBA exists.

However, CORBA is much reduced from how it was seen in the 1990s. Throughout the 1990s, the assumption was that CORBA was going to be the primary Middleware of the internet. There were two major reasons that CORBA lost its title as the primary Middleware of the internet. The main one was the fact that it came along early, and did not foresee the growth of the World Wide Web, and the corresponding growth of hacking, and the firewalls that grew up to address the hacking problem. CORBA traditionally does not work well through firewalls, as firewalls tend to focus on port 80 (and HTTP), and CORBA was designed to run directly on top of TCP/IP. The other reason CORBA lost market share was the complexity of its Application Programming Interface (API) and the fact it tried to do everything for everybody (Henning 2006), companies were unable to find sufficient well-trained CORBA programmers to meet their needs.

Let's discuss further the idea that CORBA tried to do everything for everybody: One of CORBA's main concepts was that connections between just about all combinations of programming language/ operating system/computer hardware should be possible. There are "bindings" from CORBA to many programming languages—C/C++, Java, COBOL, Fortran—the list is very long. As an example, it's completely reasonable to use CORBA to connect between a program written in Java using the omniORB

CORBA ORB (that is, the omniORB implementation of CORBA libraries (see more below) running on Linux) to a program written in C++ using the TAO ORB running on Windows. Back in the early 2000s when CORBA popularity in the industry was enormously high, I used to have students in my Client/Server course do connections between three different ORBs on two different operating systems. Remember also that CORBA was intended to connect between all different kinds of computer systems hardware: PCs, embedded systems, mainframes. So some decisions in CORBA were made to

enable the use of any kind of underlying computer hardware.

We will see this from time to time as we examine CORBA, most notably when we discuss the use of “integer” and “octet” in the Interface Description Language (IDL).

Why did CORBA end up trying to do everything for everybody? Probably because so many different entities with different needs were involved in its creation and evolution. The CORBA

specification was produced by the Object Management Group (OMG), whose membership has included/currently includes many different companies and organizations (Henning 2006). The OMG is also notable for being the organization in charge of the Unified Modeling Language (UML) specification and the Data Distribution Service (DDS) among many others.

CORBA also has another concept that was intended to reduce complexity but in some cases may have actually increased complexity: this is called *location transparency* or the idea that the actual location of an object being called doesn't matter. That is, objects that are local are treated the same as remote objects, with the result that the most complex case always occurs, that is, that local objects do not have optimized calls relative to calls to remote objects running on different hardware.

One major development in the CORBA world was the advent of the real-time CORBA specification. This allowed end-to-end predictability of how long CORBA invocations would require, an extremely important concept in a real-time/embedded systems world. To handle this, the real-time CORBA standard specified the use of object and thread priorities and predictable scheduling via threadpools, among other benefits.

Note that the word “ORB” in CORBA stands for Object Request Broker. It is used inside CORBA to refer to the CORBA libraries. It is also used colloquially to refer to different CORBA implementations by different CORBA “manufacturers” or producers, which is how I was using the term earlier when I referred to omniORB and the TAO ORB. In this textbook, we will examine implementations of the same examples in three different ORBs, the ones I've chosen are the MICO orb, omniORB, and the CORBA ORB that is included as part of the Java Development Kit (JDK)—I will call this the JDK

ORB. With MICO and omniORB, we will be looking at the C++ CORBA bindings, whereas with the JDK ORB we will look at the Java CORBA bindings.

For the code provided for the instructor with this textbook, the MICO and omniORB C++ examples were developed on Linux, while the JDK Java examples were developed on Windows. I didn't provide C++ CORBA examples for Windows because the way I've always done that in the past was through Microsoft Visual Studio, and I find every new version of Visual Studio has changed around some of the ways required to configure it to make these different ORBS compile and run. So just sticking with Linux for these wasn't so much of a moving target. In the past, I've had many CORBA ORBs running on Windows as well as Linux, however.

8.2.1 BASIC CORBA CONCEPTS

CORBA is primarily Client/Server oriented (as opposed to, say, peer-to-peer, or publish-and-subscribe) and is based on remote procedure calls to methods on remotely located objects. The client-side code contains a proxy object, this object represents the remote object in the local client space. When the Client calls a method on the proxy object, the proxy object performs all actions necessary to serialize and encode the data parameters of the method, and passes those over the underlying network (usually TCP/IP on the internet), using a special protocol called General Inter-Orb Protocol (GIOP) (which runs on top of TCP/IP) to the remote server. The remote server does the necessary processing to determine the appropriate servant code to handle the remote procedure call, and passes the data parameters to that servant. After the servant has performed appropriate processing, the response data (if any) is serialized and encoded appropriately and returned (again using GIOP, usually running on top of TCP/IP) to the Client. As you may surmise, the phrases "all actions necessary," "necessary processing," "serialized and encoded

appropriately” involve considerable work that is performed by various calls to the CORBA libraries. We will discuss these issues in the following sections.

First of all, there are two major interfaces in CORBA, the static interface and the dynamic interface. The static interface is defined at compile time, whereas the dynamic interface operates primarily at runtime. We are going to focus primarily on the static interface; however, we will also be going

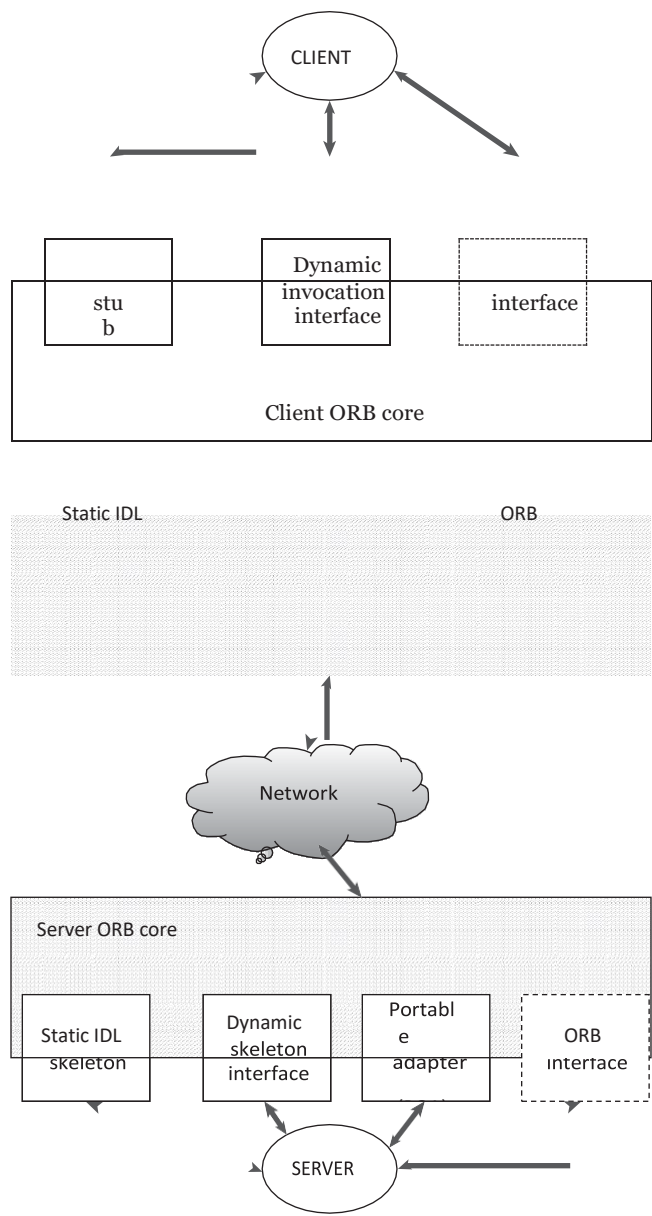


FIGURE 8.1 Client and server CORBA interconnections.

over the “any” data type, which although it can be used in a static interface is an important component in the dynamic interface.

Figure 8.1 shows a fairly common view of the connection between Client and Server using CORBA.

The interface between the Client and Server is specified using the IDL. Each implementation of CORBA (CORBA “ORB” such as omniORB or MICO) includes an IDL compiler that is used to create the static interface. The IDL compiler translates the interface that was described using the IDL into the appropriate calls to the particular implementation of the CORBA library (to CORBA libraries implemented in omniORB, MICO, or etc.) in a particular programming language (such as C++ or Java).

If you look at the client side you will see the IDL stub, whereas on the server-side is the IDL skeleton. The IDL stub is client-side code (in the appropriate programming language) that calls the CORBA library API related to Client operation. The IDL skeleton is server-side code (in the appropriate programming language) that calls the CORBA library API related to server operation. At compile time, the IDL compiler runs and creates the IDL stubs and skeletons. After the stubs and skeletons are created, the IDL stubs and the client code itself are compiled and linked together into an executable, and the IDL skeletons and the server code itself are compiled and linked together into an executable. The servant code may be linked in with the server code, or may be separately compiled and separately called by the server.

On the server-side you will also see the Object Adapter. Since CORBA 2.2, which dates from the 1998 timeframe, the Object Adapter that is used is called the Portable Object Adapter, or POA (which is pronounced POH-ah). Prior to 1998, there was a different object adapter, called the Basic Object Adapter, or BOA (pronounced BOH-ah). Unfortunately, client code written to connect to a BOA will not connect to a POA, and vice versa. Many ORBs in the past have provided separate POA and BOA implementations. However, we will focus only on the POA, since now the BOA is largely obsolete (except possibly for some remaining legacy code).

The POA is very powerful and does many server-side functions that in many other technologies would be performed by a container (as in EJBs) or perhaps by a non-standardized application server (sometimes a web server, in the case of web technologies). In CORBA, much of what a POA does, including its interfaces and the policies that control its internal behavior, are part of the CORBA standard, although some ORB manufacturers have added additional functionality. We'll discuss the POA more below, when we get to some of the code that uses it.

8.2.1.1 First Look at CORBA: Overview of Simple Echo Example

We will look very briefly at a very simple example of an echo client that sends a string to the echo server (with associated echo servant) and the servant sends that same string back to the Client. I'm using Java and the JDK ORB for this example, because for simple things CORBA code written in Java is usually easier to understand (we'll see why later on in this textbook).

To begin with, we will see the actual interface definition in a language called Interface Description Language (IDL) in Listing 8.1. Then we will see the actual Java Servant code that

implements this interface on the Server, this is shown in Listing 8.2. Then we will see pseudo-code for the Server in Listing 8.3, and pseudo-code for the Client in Listing 8.4.

Finally we will see the real code for the Server in Listing 8.5 and the real code for the Client in Listing 8.6.

Here, I'm leading into the actual server and client code bit by bit, because at this point you just simply won't understand what's going on in Listing 8.5 and Listing 8.6—there's a lot of CORBA you will have to learn first. But this will give you a feel for what is to come.

In Listing 8.1 we define a very simple remote procedure call, that passes some string in from the Client to the Server, and the Server is expected to return a copy of the string that was passed in back to the Client. The remote procedure is named “echostring.”

LISTING 8.1 Simplest Echo Interface—Actual Code

```
interface echo {  
    string echostring (in string the_echostring);  
}
```

In Listing 8.2, we see the actual Servant code in Java that will implement the interface in Listing 8.1. It may bother you that instead of implementing the interface `echo`, such as you would do in vanilla Java code (that is, in Java code that does not include CORBA), here the class `myecho` extends a class called `echoPOA`. For now at least, try not to worry about it—somewhere under the hood of CORBA implementing the interface `echo` has been taken care of.

The `echoPOA` class is something that was generated by CORBA based on the interface from Listing 8.1.

We can see what the remote procedure call `echostring` on the Servant is doing. It prints out the string that was passed in, and then it returns that same string back to the Client.

LISTING 8.2 Pseudo-Code for Servant—Actual Code

```
class myecho extends echoPOA {
    public String echostring(String the_echostring) {

        System.out.println(the_
        echostring); return
    }
}
```

Now, let's look at the general flow of execution on Server and Client, and the interaction between Server and Client, as shown in the pseudo-code in Listings 8.3 and 8.4.

For this implementation to work, you have to start the Server first. It has to be up and running and waiting for the Client before the Client can do anything.

First, on the Server, the ORB has to be initialized. There is usually some configuration information that must be read into the ORB for execution, and also various configuration choices can be made at this step. Next, on the Server, a reference to the POA is obtained. The POA on the Server is responsible for many server-side functions. One of the most important is that the POA maps the CORBA object to a Servant that implements it.

On the Server side, a CORBA object is a representation of the interface. As the Server receives remote procedure calls from the Client, these are routed to the appropriate CORBA object for that interface.

The POA “activates” a CORBA object by creating an appropriate Servant and mapping that Servant to the CORBA object such that when the CORBA object receives remote procedure calls from the Client, those are routed on to the Servant.

After the Server has created a CORBA object and activated it with an appropriate Servant, the Server writes the reference to (address of) the CORBA object to a file in a string format. This is so the Client can later retrieve the address of the CORBA

object so it can call remote procedures associated with that CORBA object.

Finally, the Server starts listening for remote procedure calls from the Client. Next, on the Client, the ORB has to be initialized. Again, there is usually some configuration information that must be read into the ORB for execution, and also various configuration choices can be made at this step.

The Client retrieves the reference (in string format) to the CORBA object on the Server side from the file where the Server previously wrote it. The Client uses this reference to create a Proxy Object. The Proxy Object represents the interface on the client side.

Finally, the Client calls a procedure (method) on the Proxy Object. This results in a remote procedure call to the CORBA object on the Server side.

The Server receives the remote procedure call from the Client, and routes this to the appropriate CORBA object. From the CORBA object, the remote procedure call is routed on to the associated Servant. The Servant performs the appropriate processing and returns the appropriate response. This goes to the CORBA object, and from there back across the communication link to the Proxy Object on the Client.

LISTING 8.3 Pseudo-Code for Server

1. Initialize the ORB
2. Get a reference to the POA
3. Create a servant object that implements the interface
4. Create a CORBA object and associate the servant object

6. Save the reference to (address of) that CORBA object in a file

LISTING 8.4 Pseudo-Code for Client

1. Initialize the ORB
2. Retrieve the reference to (address of) the CORBA object from a file (the CORBA object represents the interface on the server-side).
3. Using the reference to the CORBA object, create a

Now, compare the pseudo-code for the Server in Listing 8.3 to the actual Server code shown in Listing 8.5. Compare the pseudo-code for the Client in Listing 8.4 to the actual Client code shown in Listing 8.6. Although there's a lot you won't understand until you've read a lot more about CORBA, now that you know the overall flow from the pseudo-code you should be able to kind of see what's going on. Actually, at this point, if you keep things really simple and do cut/paste, monkey see/monkey do, you should be able to run this code and even modify it a little and play with it. This echo example, using the interface in Listing 8.1, is included in the code provided for the instructor with this textbook. I've provided this Java example that runs on the JDK ORB, but also C++ examples for the MICO ORB and for omniORB, along with instructions for how to execute them.

Since you don't really know yet what you're doing, though, if you try to do too much you *will* get yourself into a problem. CORBA is pretty complicated. Read more of the book and learn a little first before you get too deep into it.

LISTING 8.5 Simplest Echo Server

```
import
org.omg.CORBA.*;
import
org.omg.PortableSer
ver.*;
import org.omg.PortableServer.POA;

import
java.util.Properties
; import java.io.*;

...echo servant code from Listing 8.2.
```



```
// Get a reference to the POA (and POA Manager)
org.omg.CORBA.Object
the_rootPOA=orb.resolve_initial_references("RootPO
A"); POA mypoa = POAHelper.narrow(the_rootPOA);
mypoa.the_POAManager().activate();; //activate the
POA manager

// Create a servant object that
implements the interface myecho
the_echo=new myecho();

// Create a CORBA object and associate the servant
object with that CORBA object
// Get the reference to (address
of) that CORBA object byte []
servant_id =
mypoa.activate_object(the_echo)
;

// Save the reference to (address of) that
CORBA object in a file String line=null;
FileWriter thefile = new
FileWriter("echo.ior"); BufferedWriter
bufferedWriter = new
```

LISTING 8.6 Simplest Echo Client

```
import
org.omg.CORBA.*;
import java.io.*;

public class client
{
    public static void
    main(String args[])
```

```
// Retrieve the reference to (address of) the CORBA
// object from a file (the CORBA object represents
// the interface on the
// server-side). String
line=null;
FileReader thefile = new
FileReader("echo.ior"); BufferedReader
bufferedReader = new
BufferedReader(thefile);
// since only reading one value probably not
// necessary but I'm used to
// using bufferedReader around all FileReaders
line=bufferedReader.readLine();

if (line==null)
    System.out.println("no IOR found in

IOR file"); bufferedReader.close();

// Using the reference to the CORBA object, create
// a Proxy Object that represents the interface on /
// the client side
```

8.2.2 INTERFACE DESCRIPTION LANGUAGE

In this section, we examine the IDL and learn how to define interfaces to be used between CORBA clients and CORBA servers.

The “interface” keyword in these examples is used very much like the “interface” keyword is in Java: it defines a set of methods that can be called, and that must be implemented (in the case of CORBA, on the server-side), usually by a class definition. Remember that CORBA maps to many different languages, and not all those languages have built in the concept of a “class.” In C++ and Java, however, which we focus

on here, these will be implemented on the server-side by a user- defined class that inherits from and extends a CORBA library class. (The CORBA library class will be located in the stubs or skeletons that are generated by an IDL compiler.)

First, look again at Listing 8.1. In Listing 8.1, we have a very simple remote procedure call, that passes some string in from the Client to the Server, and the Server is expected to return a copy of the string that was passed in back to the Client. (Note that in practice, the fact that the name of the remote procedure is “echostring” does not in fact constrain the Server to return or echo the same string; what string is returned is totally up to the Server. However, the server code provided for the instructor with this textbook that works with this particular echo interface does, in fact, echo back the passed in string to the Client.)

Here, the data type of the parameter being passed from the Client to the Server is “string.” This could be many different data types, however. If you look at Listing 8.7 you will see the data type of the parameter is “short.”

The IDL uses “short” to mean a 16-bit integer, and “long” to mean a 32-bit integer. In the CORBA IDL, we also have “unsigned short” and “unsigned long.” So a “short” is a signed 16-bit integer, and a “long” is a signed 32-bit integer. This means that you could send, say, a -1 in a “short” but not in an “unsigned short,” which would only allow you to send a +1. (It also means that since you’re not using it to store negative numbers, an “unsigned short” can contain (almost) twice as many positive numbers as a “short.” I use the word “almost” here since zero is considered a positive number. Think about it in hexadecimal and binary and work it out. Nice little exercise for the student. You can do it!). The IDL also supports “long long” and “unsigned long long” for 64 bit numbers.

The CORBA standard did not use the keyword “integer” in the IDL because the word “integer” has meant different things to different computer hardware/operating systems in the history of computers. Traditionally, “integer” has been a data type associated with a non-signed value that is stored in one word of the computer hardware. And in the history of computing, the size of a “word” has varied quite a bit. Some old microprocessors had words that were 8 bits wide. Other microprocessors have had word sizes that were 16 bits, 32 bits, 64 bits. Some old mainframe computers, the Control Data Cyber computers, had 60-bit wide words. (Note that the number 60, although an even number is not a power of 2. In the old days, that obviously wasn’t considered a necessary requirement for word size). As an interesting tidbit of information for you, the first computer I ever programmed on was a CDC Cyber computer.

The IDL uses the word “octet” to refer to 8 bits. Why didn’t the CDC use the word “byte” for 8 bits? Well, in the old days, like on the CDC Cyber computers, a byte was not 8 bits. According to Wikipedia, the CDC 6600 used a 12-bit byte and a 6-bit character. My own memory is that the CDC Cyber computer I programmed on back in undergraduate school at Georgia Tech used 6-bit bytes, which is what I’ve been telling students for years. Maybe the Wikipedia article is correct, however, it’s been a while. In any case, we see here that the word “integer” and the word “byte” could not be used by any IDL that intended to target old mainframe computers and such. So we can see some of the choices made in the CORBA standard.

Perhaps oddly, CORBA didn’t make some restrictions in the IDL that you might have expected: for example, it’s perfectly acceptable to use the word “if” and “while” and “switch” and “for” as names. It would be legal to have the following:

```
interface if {  
  
    while switch (in for the_data);  
  
}
```

Of course, this wouldn’t compile when the IDL compiler tries to convert it to C++ or Java, but in other languages, perhaps. ☺ Yet another way in which CORBA was trying to be everything to everybody.

Consider Listing 8.1 again. The word “in” here means that the data is going *from* Client to Server, but not back from Server to Client. That is, the Server cannot change the data in the_echostring.

In CORBA, the direction from Client to Server is *in*, and from Server to Client is *out*. So if we wanted to send another data item back (in addition to the string that is the return type of the function), we could make it an *out* data type. The IDL also supports *inout* data types, an example is shown in Listing 8.7.

With an *inout* data type, the Client can send a value to the Server in the data item of type *inout*, and the Server can also send a value back to the Client.

Note that CORBA assumes IDL strings are null terminated. See Section 8.2.7.1 for a discussion of how IDL strings are stored as they are being transmitted to/from Client and server.

LISTING 8.7 Echo Interface Using Data Type Short

```
interface echo {
    string echoshort (inout short the_echoshort);
}
```

LISTING 8.8 Echo Interface with inout

```
interface echo {
    string echostring (inout string the_echostring);
}
```

In the IDL, you can also group different interfaces into namespaces (called “modules”). If you have a complex set of interfaces, that allows you to organize the interfaces for better understanding. It also allows you to reuse interface names in different modules. If you want to access these interfaces inside the modules, however, you now have to use the whole name for an interface, for example, the one in Listing 8.9 would be referred to as: `echomodule.echo`. Or alternately, for example, in C++, you could say “using namespace echomodule” at the beginning of your code.

LISTING 8.9 Echo Interface with Namespace

```
module
    echomodule
    { interface
        echo {
            string echoshort (inout short the_echoshort);
        }
    }
```

CORBA also provides numerous built-in system exceptions, or you can define user exceptions of your own. Listing 8.10 shows a user-defined exception named “Bad_Message.”

There is a fairly long list of CORBA-provided system exceptions. Some of the more commonly seen of these are `COMM_FAILURE` (this usually means the ORB could not set up basic communication over the network to the other side or else loses it), `MARSHAL` (this means the request or response is not correct; in my own experience this often means that one of the ORBs is not working correctly, perhaps it was not compiled correctly), `BAD_PARAM` (data parameter out of range), `OBJECT_NOT_EXIST` (the CORBA object on the server-side is not there), or the very common (in my own experience) `TRANSIENT` exception. `TRANSIENT` means the ORB could not find the object but doesn't know why, and for all it knows it might be a temporary condition. `TRANSIENT` is often thrown when the client side does a *narrow* and the Server is not running. But we'll see this again later.

Unfortunately, the `UNKNOWN` system exception is also common. This means it was some exception that the CORBA implementation you're using doesn't understand. Usually it's a non-CORBA exception.

LISTING 8.10 Echo Interface with Exceptions

```
interface echo {  
    exception Bad_Message{};  
    string echostring (in string the_echostring) raises
```

Now we're going to expand beyond the echo example, and look at various interface examples that are transferring slightly more complicated data—that therefore we don't really want to echo back. ☺

In Listing 8.11, we see a user-defined struct being passed in the interface, as a parameter in a remote procedure call. Here the struct is named “theData” and we define a new data type called StructType. Then we define a parameter in the remote procedure call as a struct of type StructType.

LISTING 8.11 Interface Sending a Struct

```
struct  
    theData {  
        long  
        firstvalue;  
        string secondvalue;  
    };  
typedef theData
```

In Listing 8.12, we see an interface that is returning a struct as the return value of a remote procedure call. Here the struct and the data type StructType are the same as in Listing 8.11.

LISTING 8.12 Interface Returning a Struct

```
struct
    theData {
        long
        firstvalue;
        string secondvalue;
    };
typedef theData StructType;
```

In Listing 8.13, we see a user defined array being passed as a parameter in a remote procedure call. In CORBA, the whole array will be sent. That is, if you have an array on the Client of length 20, but you only have set 3 data values in that whole array, it's going to send those 3 values but it's also going to send 17 other empty values (maybe zero, maybe who knows what, depends on whether you've initialized the array or not).

One of the *gotcha* aspects of CORBA is, you have to remember that when CORBA sends an array, it doesn't say anything about whether the array index begins at 0 or at 1 (or for that matter at any other number). Remember that CORBA is intended to connect practically any arbitrary language to any other arbitrary language. So you can never make any assumptions about how the array indices will be used at the other end.

LISTING 8.13 Interface Sending an *in* Array

```
typedef short  
ArrayType[20];  
const short  
MAX=20;
```

In Listing 8.14, we see a sequence being sent as a parameter in a remote procedure call. Sequences in CORBA transmit only the data that was actually set in the sequence. That is, a sequence can be up to a certain size (perhaps 20 as in the earlier array example), but if you have only 3 elements set in the sequence, only 3 will actually be sent. This is very useful in the case of sparse data (that is, data with lots of holes in a data structure and only a few data items).

Here we are sending a sequence of long (that is, 32-bit integer) values. If we wanted to send a sequence of, say, floats (that is, IEEE single-precision floating point, 32 bits), we would do instead:

```
typedef sequence<float> SequenceType;
```

whereas if we wanted to send doubles (that is, IEEE double-precision floating point, 64 bits), we would do:

```
typedef sequence<double> SequenceType;
```

For whatever IDL mapping you're using, there will be a

length() function attached to a sequence that will allow you to find out how long the sequence is. We'll look at this later in the C++/C mapping.

LISTING 8.14 Interface Sending a Sequence

```
typedef sequence<long>  
SequenceType; interface  
sending_stuff {
```

LISTING 8.15 Interface Using an Attribute

```
interface  
    sending_stuff {
```

Next, we will discuss the “attribute” keyword in IDL. We see an example of this in Listing 8.15. I thought about *not* discussing the “attribute” keyword because there are always a few students who don’t listen and therefore don’t understand the “attribute” keyword but try to use it incorrectly and cause themselves a lot of grief. But lots of CORBA code out there in the wild uses it, from what I’ve seen, so I finally decided I better discuss it.

Before I do, though, let’s review what CORBA *is* and what CORBA *is not*. CORBA *is* a way to send data from one computer to another computer across a network. When you use CORBA, you’re generally pretending that the Client is connecting direct to the Server, that is, since you’re ignoring the lower networking layers (as in most middlewares), and that therefore you don’t have to know how that data is being routed between any intermediate computers on the network. That is, most of the time you’re only worried about the end-to-end connection.

Think about what a network is. The simplest network would be a wire stretching from one computer to another. Well, obviously our networks that we’re going to use are a lot more complicated than that. But when we’re doing CORBA, for the most part we can think about our network as being as simple as that wire. Because the main purpose of middleware is to let us ignore the network layers below the mid- dleware. And because CORBA is usually considering only the endpoints (Client and Server).

So it’s reasonable in a large sense to *pretend* for the sake of an example that CORBA is working from one computer to another computer across one wire.

Now let’s look at that wire. It’s just a wire, right? I can send bits of data moving across that wire, from one end to another, by changing the voltage on the wire. That is, at one end I raise the voltage levels and lower the voltage levels. A high-voltage level might be a logic 1 and a low voltage level might be a logic

o. At the other end, I look at what voltage levels are there to determine which bits have been sent. But the voltage levels are only temporary, so I have to look quick because *they will go away*.

Whew, this is an oversimplification. ☺ But I'm going to make an important point using this super simple example. Here comes the point:

- I don't see *anywhere* on that wire that I can actually *store* any data, do you?
- I mean, I didn't put any flip flops on the wire. I didn't put any registers of any kind on the wire. For those of you who might have an electrical background, I didn't put any capacitors on the wire.
- SO THERE IS NO WAY TO STORE ANY DATA AT ALL ON THIS WIRE.

Let me say that again: There is no way to store any data on the wire.

Reasoning toward the larger world: You don't store data on network connections! You might store data at one end of a network, or possibly at an intermediate point in a network. But on the network connections themselves, no, it's just a collection of wires, or maybe radio or laser signals, sending data from one point to another point.

But in CORBA we're pretending we have an end-to-end, Client to Server, connection, and we're ignoring any intermediate points on the network. So there is NO DATA STORAGE in the network interface.

So the "attribute" keyword in the IDL *cannot possibly* be used to mean any data is stored, or that any kind of data variable is being defined.

What the "attribute" keyword *is*, it's a shorthand notation to signify getting and setting a particular variable at the servant. That is, it defines a *set* routine that allows the Client to send a value to be written to a particular variable on the Servant, and it defines a *get* routine that allows the Client to request that the Servant send it back the value of a particular value.

For example, the attribute `the_value` in the interface in Listing 8.15 would result in two remote procedure definitions:

1. A remote procedure named the_value with no parameters passed, that returns a Short value
2. A remote procedure named the_value with a Short value passed as a parameter, that does not return any values (has a void value as the return value)

If we look at the C++ code using either omniORB or the MICO ORB that would implement these remote procedure calls, they look like this:

- `virtual CORBA::Short the_value();`
- `virtual void the_value(CORBA::Short the_new_value);`

So now you know what an attribute is. *Please* don't try to store any data in your IDL !!!

Now let's look at the "any" data type. Listing 8.16 shows an "any" data type. The "any" data type is just what it says, using this data type you can send any kind of data. You could write a float value into an "any" on the Client and send it as the mydata parameter to the sendvalue method in Listing 8.16. You could write a string value into an "any" on the Client and send it as the mydata parameter to the sendvalue method in Listing 8.16.

The way this works, an "any" data value contains a TypeCode value that specifies what kind of data is stored in the "any." Let's say that a Client is sending an "any" to the Server. The Server can extract the TypeCode from the "any" and then the Server will know what kind of data has been sent and can handle it appropriately.

Note, however, if the Client and Server have previously agreed on the type of data that will be sent (perhaps for a period of time?) then the Server has easier ways to extract the data type from the "any" in which the Server does not have to check the TypeCode. We will see an example of this later on.

However, we're not going to get into handling "any" with any depth. That's because probably the most common use of an "any" data type is in the dynamic interfaces, which we're only going to look at briefly.

There is one thing to note about TypeCodes. CORBA exceptions have an associated TypeCode, that you can check if you need to, to determine what kind of exception has occurred.

LISTING 8.16 Interface Sending an Any

```
interface sending_stuff {  
    string sendvalue (in any mydata);  
}
```

A few other leftover IDL items:

First the CORBA IDL enumerated type called “enum” looks like this:

```
enum Money {euro, dollar, pound, yen};
```

which of course should remind you of C++/C enums.

One thing to remember about IDL enums: You don’t know which actual numbers will be mapped to the values—for example, “euro” above might be mapped to the number 0 or it might be mapped to the number 99. You can’t control it and you shouldn’t make any assumptions about the numerical values.

There’s a thing in IDL called “discriminated unions.” It allows you to swap at runtime from sending one kind of data to sending another kind of data, based on a discriminator value that allows you to swap from sending one kind of data to sending a different kind of data. It’s supposed to be more efficient than sending all the different kinds of data you need to send every time, when you only need to send one kind of data at any one time. It’s supposed to be easier and cleaner to understand than using an “any” data type. Well, I believe it is more efficient but as far as being easier to understand

than an “any,” that has not been my experience with students trying to learn this in the past. I’m not giving an example here because I think you should avoid it, ☺ at least while you’re a CORBA beginner. Later on, once you know what you’re doing, you can look it up, and use it if you really feel you need it.

8.2.3 CORBA IDL TO C++/C AND CORBA IDL TO JAVA BINDINGS

When the IDL compiler creates stubs and skeletons, what it is really doing is automated code generation. The stubs and the skeletons are code in whatever language you’re using—C, C++, Java, Python, could be many different languages—that implement various calls to the CORBA libraries (the CORBA libraries themselves have been compiled as part of the ORB when it compiled, and are located in a directory somewhere; you’ll have to link to that directory).

The stubs and skeletons implement various definitions that you use in your own code. For example, in the MICO ORB in the `echo.h` that defines the skeleton (that is, server-side) file that was created using the IDL compiler on the `echo` example, you would have the following definition that is part of the interface implementation between your code and the MICO ORB:

```
class POA_echo : virtual public PortableServer::StaticImplementation
...

```

whereas in omniORB in the `echo.hh` file that was created using its IDL compiler, you would have:

```
class POA_echo :
    public virtual _impl_echo,
    public virtual ::PortableServer::ServantBase
...

```

So as you see, somewhat different implementations but leading to the same CORBA class to be used to implement an interface.

The servant class you use that actually implements the interface, would begin like this:

```
class myecho : public virtual POA_echo
```

...

for both the MICO ORB and omniORB.

In Java this of course looks a little different. Using the JDK ORB on a similar echo example, you have a file named echoPOA.java that is part of the skeleton (server-side) code. The part of the implementation of the echo interface in this skeleton looks like this:

```
public abstract class echoPOA extends org.omg.PortableServer.Servant  
    implements echoOperations, org.omg.CORBA.portable.InvokeHandler
```

...

and your servant class that actually implements the interface would look like this:

```
class myecho extends echoPOA
```

...

But again, you're using a very similar CORBA class to implement the interface to what you did in the C++ ORBs.

In all these cases, very similar stuff, obeying the same rules. It's just a little different because in these cases, different people implemented it in a different way, and in two different languages.

So these are some of the things that can be seen when talking about different ORB implementations. And now we're also beginning to look at different bindings to different languages.

When we consider the different language bindings, there are some major differences between different languages. They all generally have to do the same calling-CORBA-libraries kinds of things, but each language has its own peculiarities that you have to learn.

When the CORBA standard was originally created in 1991, the very first language mapped to was C. Yes, C, not C++, the mapping to C++ didn't occur until 1996. The Java language mapping didn't happen until 1998. (Java itself was released in the 1995 timeframe.)

When we compare the C++/C binding to the Java binding, the biggest difference between them is that Java does garbage collection and C++/C does not. So the CORBA standard tried to help with C++/C memory management, and the C++/C binding acquired a lot of—useful, interesting, annoying (depending on your viewpoint)—features related to memory management.

Have you ever heard of a bug called a memory leak? In my experience it is the most aggravating and difficult kind of bug to catch of all bugs. Fixing it, however, is usually a breeze—if you're using C++ usually you just have to add one more “delete” or “free” statement, somewhere in your code. It's finding the appropriate “somewhere” that is the difficult part.

What happens in a memory leak is, somehow or other when you're done with using dynamically allocated memory, you

forgot to return that memory to the list of free memory that can be reused (heap). Even if this is an infrequent bug, eventually all the free memory on your system will be used up, and your system will crash! But the memory leak bug that caused the issue may have actually happened a very long time before—perhaps hours or days. All you will be able to see when you look at the immediate cause of the bug is that your code tried to allocate new memory, but there was no free memory available, so the system crashed.

Back in the 1980s, I used to work at a modem company. We had a major customer who bought thousands of our modems and mounted them in racks. What the customer found was, if they didn't reboot all those thousands of modems at least once every week, then they would crash in different ways at different times. I and one other software engineer tracked that bug for weeks, all the time with a major customer breathing down our necks and being checked on repeatedly by important managers. We finally set up test modems to trap various bug possibilities, and any time a combination of circumstances happened, we started a modem light blinking (so the customer could tell this situation had happened), and we wrote the contents of the modem memory, in hexadecimal, out an I/O port. The customer captured this data and sent it back to us as very long printouts. Then we spent weeks examining this hexadecimal data. When we finally found the bug, it was a one-line fix.

Java has a wonderful thing called garbage collection that fixes this kind of thing. Every so often, Java runs through and returns all unused memory to the heap (free memory list). There can be some performance negatives versus C++/C involved, the arguments pro and con are lengthy.

The CORBA C++ bindings attempted to make memory management harder to forget and largely required. In my opinion, they achieved this, but at the cost of making the code much harder to understand, that is, it kind of looks like a new

language and not C++ at all. Sometimes (especially in the past when CORBA was the hottest technology around) I've had my students do a CORBA program in C++ and get it working there, then re-do the same CORBA program in Java. Invariably the students thought the Java program was easier. That's because CORBA in Java didn't have to add a lot to the Java language to do memory management.

We will look at the CORBA C++ binding next, after that at the CORBA Java binding, so we can compare the two.

8.2.3.1 CORBA IDL to C++ Binding

The CORBA C++ binding maps IDL basic types to C++ using a CORBA namespace. That is, the type “float” in IDL would show up as “CORBA::float” in C++. The type “boolean” in IDL would show up as “CORBA::Boolean” in C++, while an unsigned short in IDL would show up in C++ as “CORBA::UShort” and an unsigned long long in IDL (64 bits) would be “CORBA::ULongLong.”

You get the idea: for the basic types add “CORBA::” in front of the IDL name, then capitalize the first letter of each instance of the data type. The IDL “unsigned” becomes “U” in front of the data type.

Arrays in IDL are mapped to C++ arrays and have the same name as in the IDL (they’re user- defined types so you don’t add “CORBA::” in front). For example, the servant code in Listing

8.17 implements the IDL interface previously shown in Listing 8.13. (By the way, ignore the string versus char * stuff for now, we’ll talk about this later.)

LISTING 8.17 CORBA IDL *in* Array Servant in C++ Example

```
class myreceiver : public virtual POA_sending_stuff
{
public:
    virtual char * sendvalue (const ArrayType myarray,
                             const short size);
};

char * myreceiver::sendvalue (const ArrayType myarray,
                              const short size)
{
    int the_real_size;

    the_real_size=size;
    if (the_real_size >
        MAX)
        the_real_size=MA
    }
}
```

The array is defined in the parameter “ArrayType myarray” and myarray is accessed similarly to any other C++ array:

```
myarray[i]
```

The IDL sequence from Listing 8.14 would map to servant code such as that shown in Listing 8.18.

LISTING 8.18 CORBA Sequence in C++ Example

```
class myreceiver : public virtual POA_sending_stuff
{
public:
    virtual char * sendvalue (const SequenceType
        &mysequence);
};

char * myreceiver::sendvalue (const SequenceType
    &mysequence)
{
    for(CORBA::ULong i=0;i<mysequence.length()+1;i)
```

As you can see, the sequence is accessed the same way any other C++ sequence is accessed: `mysequence[i]`.

If you want to find out how long the sequence currently is, you can do:

```
CORBA::ULong the_length=mysequence.length();
```

You can set the length of a sequence forcibly by doing:

```
mysequence.length(15);
```

or:

```
mysequence.length(2);
```

That is, you can extend the length or truncate the length (and thereby drop off the last few characters).

You can find the current maximum length of a sequence by doing something like:

```
CORBA::ULong the_max=mysequence.maximum();
```

One tricky thing in the CORBA C++ binding is that an IDL string maps to: `char *`. When you allocate a new string, you don't use `new` or `delete`. There are special functions in the namespace `CORBA` that allow string manipulation: `CORBA::string_dup` duplicates a string, `CORBA::string_alloc (ULong length)` allocates a string of the given length, and `CORBA::string_free (char *)` deallocates the specified string.

This was done, according to Schmidt and Vinoski (2000) partly because when the IDL to C++ mapping was created back in 1994, C++ did not contain a string data type. Also, the CORBA IDL to C binding mapped "string" to "`char *`." Additionally, there were some inter-language call issues, and one particular company (IBM) required the IDL string to "`char *`" mapping.

The easiest way to allocate a string and initialize it at the same time is:

```
char * mystring = new CORBA::string_dup("ThisIsTheBeginning");
```

Then you can explicitly free it, if you want to, using:

```
CORBA::string_free(mystring);
```

However, there's another way to handle a string that does automatic memory management of the string. That is to use an “_var” type. When you use “_var” attached to a particular data type, you're telling CORBA to do some memory management. Here's a very short example, in Listing 8.19.

LISTING 8.19 CORBA _var Example

```
int i;
(...i is read in from
somewhere...) if
(i==4)
{ // this is the beginning curly bracket
  CORBA::String_var mystring = new
```

Note that we have turned our original `char *` string into a string of type `CORBA::String_var`. Do you see the curly brackets? See that we have defined `mystring` *inside* these curly brackets. That means that the *scope* or lifetime of `mystring` is the lifetime of the curly brackets. Because we used an `_var` type here, when the code exits the curly brackets, the memory containing the “ThisIsTheBeginning” is returned to the heap (list of free memory) for use by other programs. (We're assuming in this case that “...do some other stuff...” didn't change the contents of `mystring`.)

The `String_var` is a class that is defined in the `CORBA` namespace. It is a *wrapper* around “`char *`.” So the “ThisIsTheBeginning” is pointed to by a pointer of type `char *` that is defined *inside* the `String_var` class.

You can also wrap an IDL “any” data type with a `_var`

(CORBA::Any_var), and you can similarly wrap the user defined C++ types: struct, class, or union. Additionally, you can wrap an array or a sequence with a _var type. For example, let's look at the following IDL for a struct:

```
struct thestruct {  
    short value;  
};
```

In the associated C++ mapping, you could use:

```
thestruct_var  
mystruct; mystruct-  
>value = 10;
```

Note that you *don't* do "CORBA::thestruct_var." You just do "thestruct_var." That's because it's a user defined type, and not one of the CORBA basic types that's built in to CORBA. Your wrapper class that creates the _var type based on the struct is generated by your ORB's IDL compiler, and it's defined in your stub/skeleton.

Note that you *must* access the items in the struct like this:

```
mystruct->value=10;
```

You *cannot* use:

```
mystruct.value=10; //WRONG
```

That's because the automatically generated wrapper class just works that way.

There are methods defined in all `_var` wrapper classes that access the wrapped data type directly. Three of these (`in()`, `inout()`, and `out()`) can be used to help you in passing `_var` data types as parameters to a function. In one case, the method `_retn()` can be used to return the wrapped data type as a return value from a function.

For example, in a `CORBA::String_var` class, to get at the `char * string` inside the `String_var` class for the purpose of using the `char * string` as a parameter in a function call, you could do one of three different things, depending on your needs:

Consider the following `String_var` variable:

```
CORBA::String_var mystring=CORBA::string_dup("Hello World");
```

- To pass `mystring` into a function that will not change `mystring`, that looks like this:

```
void myfunc(const char * thestring)
```

you could call it as follows:

```
myfunc( mystring.in() );
```

- To pass `mystring` to a function that could change `mystring` and return it to the calling program, in that case the function to call looks like this:

```
myfunc( char * & thestring)
```

and you could call it as follows:

```
myfunc( mystring.inout() );
```

- To pass mystring to a function that could use mystring, but then will deallocate its memory and *not* return the mystring value to the calling program, in that case the function to call looks like this:

```
myfunc( char * & thestring)
```

and you could call it as follows:

```
myfunc( mystring.out() );
```

Now let's assume that we have a function defined as follows:

```
char * myfunc()  
{  
...  
}
```

```
CORBA::String_var mystring=CORBA::string_dup("Hello World");

...

}
```

to return the string wrapped by the `String_var` `mystring`, we would do:

```
return mystring.retn();
```

so the function now would look like this:

```
char * myfunc()
{
...
CORBA::String_var mystring=CORBA::string_dup("Hello World");
return mystring.retn();
}
```

I should note that you could also pass a parameter such as `CORBA::String_var` in `mystring` by defining your array parameter as a `String_var`, such as this:

```
void myfunc(String_var the string)
```

However, this is *not* a good idea as it is very inefficient: a new `String_var` object is created every time this function is called. Just how inefficient this would be depends on the ORB implementation, but remember that when you're using `String_var` you're telling CORBA to do your memory management for you: Think about it. ☺

Another *gotcha* in CORBA: It's generally a bad idea to mix C++/C style pointers and `_var` variables pointing to each other/storing the same data because every time your code exits a pair of curly brackets (exits scope), your `_var` variable will

have its memory returned. If you've got any other C++ pointers pointing to that same `_var` variable, or to the contents (the wrapped data) inside that `_var` variable, after the `_var` variable exits the pair of curly brackets your pointers will be pointing to garbage. That is, your C++/C pointers will be pointing to memory on the heap, that some other program could have used for something. Basic way to fix this: Don't have C++/C style pointers point to `_var` variables.

Also, when I first started doing CORBA stuff, I was working for a U.S. Army contract where the army wanted me to determine the degree to which CORBA was interoperable. That is, their primary concern was, if they were going to use CORBA, then they wanted to be able to swap ORB vendors at any time; they didn't want to be stuck with one vendor. There are two aspects of this: One, the CORBA code needs to be able to *connect* with any given CORBA ORB, and two, it should be possible to transfer your CORBA code from any ORB to another (at least in the same programming language).

When I started working on this contract, it seemed to me originally that the best way to do this was to use as little CORBA style code (such as `_var` constructs and `_ptr` constructs—we haven't talked about `_ptr` yet, we'll see that later) as possible, and use vanilla C++/C wherever I could. This turned out to be largely a bad idea. A lot of C++/C CORBA ORBS just plain refused to work (in certain places) unless I was using the CORBA constructs when I was accessing the ORB. So this was just a bad idea and I shouldn't have gone down that path in the first place—if you're using a standard CORBA library, for goodness' sake, use the standard CORBA library.

I mention this here because this is the very thing that I've found a lot of my students have tried to do over the years. Believe me, I understand the reasoning: you the student already know C++/C and

this CORBA C++/C language mapping is tough to understand so you think maybe you can bypass it. Take it from me, it can be done to a degree but not completely, and the effort is much worse doing it this way. And your code looks weird and is hard to debug. So *don't* do that. (On the positive side, the main place you end up using `_var` variables is with return values and out parameters, where you are looking at possible memory leaks. So you don't necessarily have to use these everywhere in your code. Be careful and try to keep your true C++/C code separate from your CORBA C++/C binding code.)

The TAO ORB (TAO stands for The ACE ORB, so saying "TAO ORB" is like saying "The ACE ORB ORB," but you understand what I'm saying so who cares if it's a bit silly) especially gave me grief when I was trying to do this. The TAO ORB is based on The Adaptive Communication Environment (ACE) which is an object-oriented framework that implements various communication patterns and is quite powerful. Back when I was using TAO in the early 2000s, ACE was implemented in a macro language (I assume it still is, I haven't looked recently). I spent a lot of time converting ACE macros to standard C++/C just to prove TAO would work without those macros themselves actually being used. There were a few places where this just wasn't possible, the ORB wouldn't compile without the appropriate macro. Bottom line: If you're using TAO then I recommend using ACE like you are supposed to.

As part of this `_var` discussion, we've seen parameter passing now using `CORBA::String_var` as an example. Passing other data types as parameters to functions is similar: for "in" data types this maps to "const" parameters, for "inout" and "out" parameters to functions, you would use "datatype&." For example, to do a struct as either an "in" or an "inout" parameter you would use "struct &" and to do an enum as either an "in" or an "inout" parameter you would use an "enum &."

Note that the array data type is different. When you use an

array data type as an “in” parameter, it would be “const arraytype,” similar to what you did with any other “in” parameter. However, for “inout” or “out” you would say “array_slice *.” For return type you would also use “array_slice *.”

For return types for sequence, union, and struct, you would use “sequence *,” “union *,” and “struct *,” respectively.

There’s another CORBA C++ binding item that is important to talk about: `_ptr`. The “`_ptr`” type acts a lot like a pointer to a C++ class. It’s normally used on the client side to point to a proxy object (the proxy object represents the servant in the address space of the Client, we’ll talk about this more later on). You can also use a `_var` type to point to the proxy; it works similarly to the `_ptr` but it also does memory management of the proxy (deletes the proxy object when the code leaves the immediate curly bracket scope).

The way the `_ptr` operates is as follows, let’s assume we’re calling an `echostring` type method on the remote servant. This is part of the client code we’d use to call the servant, when we’re using the interface previously shown in Listing 8.1 (here the class “echo” is the proxy class):

```
echo_ptr myecho = do a lot of stuff to create a proxy that is connected to the
remote if (CORBA::is_nil (myecho)) {

    cout << “the server is missing” << endl;
    exit (1);

}
```

Sometime in the future you would have to do:

```
CORBA::release(myecho);
```

to return the memory used by that echo object to the heap (list of unused memory).

Or you could use `echo_var` instead of `echo_ptr`:

```
echo_var myecho = do a lot of stuff to create a proxy that is connected to the remote.
```

The “is_nil()” method that is automatically generated by the IDL compiler to check the `_ptr` (or `_var`) that points to the proxy can be used to see if the `echo_ptr` (or `echo_var`) has been initialized or not.

When you’re using `_ptr`, it keeps a reference count of the number of instances of “echo” being pointed to by the `echo_ptr`. For example, there’s a method called `_duplicate` that could be used to make a copy of the original `echo_ptr`. In that case, there would be two (or more, depending on the number of calls to `_duplicate`) that point to the same proxy object. So in this case, `CORBA::release` would have to be called for the original proxy object, plus once for every time `_duplicate` had been previously called, before the echo object itself is returned to the heap.

Let’s take a brief look at the Any data type and how it can be handled in the CORBA to C++ binding. We originally saw an IDL that used Any back in Listing 8.16. Let’s say, for example, that we want to send data of type “long”, then on the Client we could do:

```
long the_real_value;

the_real_value=1934;

CORBA::Any myvalue; myvalue<<=(CORBA::Long)
the_real_value;
```

Then on the servant if we *knew* the data we were getting was of type “long”, then we could do:

```
CORBA::Long myvalue=0;

mydata >>= myvalue;
```

I’m ignoring the situation where we *don’t* know what the data type is, and have to extract it from the `TypeCode` in the `Any`. That’s getting away from the static interfaces and more into the dynamic interfaces, and I’m largely ignoring those for this

(introductory) textbook.

8.2.3.2 CORBA IDL to Java Binding

The CORBA IDL to Java language binding is in many ways easier to understand than the CORBA IDL to C++/C binding: Since Java does garbage collection, there was no need to memory management to the language. This simplified basic CORBA interactions—there are no `_var` types and there are no `_ptr` types in the Java binding. For the most part, CORBA IDL types map directly to Java types.

For example, a module in CORBA IDL maps to a Java package. A boolean maps to a `Boolean`, an octet maps to a byte. The IDL types *short* and *unsigned short* map to a Java `short`. An octet maps to a Java byte. A *long* and *unsigned long* map to a Java `int`. A *long long* and an *unsigned long long* map to a Java `long`. *Float* maps to *float* and *double* maps to *double*.

Very conveniently, a *string* maps to `java.lang.String`. IDL *sequences* and *arrays* both map to Java arrays. IDL *enums*, *structs*, *unions*, and *exceptions* map to Java classes.

The IDL *Any data type* maps to `org.omg.CORBA.Any`.

Probably, the most complicated-looking part of the CORBA IDL to Java mapping are the *Helper* and *Holder* classes. There are *Helper* and *Holder* classes for the basic data types defined in the CORBA libraries. For any user-defined types, the IDL compiler generates a *Helper* class and a *Holder* class for each one.

The *Helper* classes allow reads and writes to CORBA I/O streams (`org.omg.CORBA.portable.InputStream` and `org.omg.CORBA.portable.OutputStream`) and inserting/extracting values from type *Any data*. (The `org.omg.CORBA.portable.InputStream` and `org.omg.CORBA.portable`.

`OutputStream` are the Java mapping classes for marshaling data to and from the format (Common Data Representation [CDR]) that CORBA converts data into for transmission between computers. We'll see more about this later on.)

The `Helper` class also implements the *narrow* routine, among other utility routines. We'll look at the *narrow* routine later on, but among other things it does a type downcast of a general CORBA object to a particular interface type.

For both the interface shown in Listing 8.13, and the interface shown in Listing 8.20, the Client would call the remote as follows):

```
sending_stuff proxy = sending_stuffHelper.narrow(obj);
```

This is one example of a `Helper` class, in this case calling a *narrow* (after creating a proxy object, we'll discuss that later).

The `Holder` classes are used when there is an *inout* or an *out* parameter to an operation in the IDL. Method parameters in Java are implemented as pass-by-value; that means a copy of the variable is passed in the parameter rather than the parameter itself. So if you need a pass-by-reference (as when you're using an *in* or *inout* parameter), that is, when you want the routine you're calling to be able to change the value—in the calling program—of that variable you're passing, you have to do something else. What a `Holder` class does is, it's a wrapper around the data type you're wanting to pass- by-reference. Then you change the contents of the `Holder` class. `Holder` classes pass data to the `Helper` classes to be read or written.

An IDL interface maps to: `Operations`, `Helper`, and `Holder` classes. The *operations* class simply provides the definitions of the remote procedure calls that are part of the IDL interface.

In Listing 8.20, we have the Java code for the servant that implements the interface previously shown in Listing 8.13. (Compare this to the C++ servant shown in Listing 8.17 that

implements the same interface.) This illustrates an array passed as an *in* parameter. It also shows how the IDL to Java binding handles IDL constants and typedefs.

LISTING 8.20 CORBA IDL *in* Array in Java Example

```
public class myreceiver extends sending_stuffPOA
{
    public String sendvalue (short[] myarray, short size)
    {
        int
        the_real_size
        ;
        the_real_size
        =size;
        if (the_real_size >
            MAX.value)
            the_real_size=MAX
```

Here we are passing an array as an *in* parameter, so no Holder classes are used. Note we use the underlying element type rather than the type name (in this case, `ArrayType`) from the typedef

used in the IDL. This is because Java as a language does not contain a “typedef” language construct. (A typedef could also be implemented by a Helper class.)

Note that the array is accessed just like all Java arrays:

```
myarray[i]
```

Also, in Java, to access an IDL const, we use (the constant’s name).value. This is because Java as a language does not contain a “constant” language construct. So what the IDL mapping to Java does, it creates an interface with the name of the constant (in the code included with this textbook, this is contained in the file Max.java):

```
public interface MAX
{
    public static final short value = (short)(20);
}
```

which has a public static value that contains the constant value. You get at this value by using (the interface’s name).value.

Now in Listing 8.21, we modify the previous IDL array example from Listing 8.13 to now send an array as an *inout* parameter.

LISTING 8.21 Interface Sending an *inout* Array

```
typedef long
ArrayType[20];
const long MAX=20;

interface sending_stuff {
```

In Listing 8.22, we have the Java code for the servant that implements the interface in Listing 8.21.

Here, since we have sent an array as an *inout* parameter, we must use a Holder class, in this case

“ArrayTypeHolder” is the name of the class.

Because we are sending the array as an *inout* parameter, that says we are supposed to modify the array inside the servant. So this servant multiplies each array value by 10. Just for something to do. ☺

LISTING 8.22 CORBA IDL *inout* Array in Java Example

```
public class myreceiver extends sending_stuffPOA
{
    public String sendvalue (ArrayTypeHolder myarray, int
    size)
    {
        int
        the_real_size
```

```
System.out.println("Original value of myarray["+i+"]  
is "+myarray.value[i]);  
myarray.value[i]=myarray.value[i]*10; // Multiply  
each myarray value times 10 System.out.println("Value  
to pass back to client of myarray["+i+"] is  
"+myarray.value[i]);
```

To access an item in an Object of type Holder class, you use “.value” to talk direct to the item, then you access it as usual. In this case, we’re talking to an array, so we access it by using square brackets:

```
myarray.value[i]
```

As we previously did with the CORBA IDL to C++ binding, let’s take a brief look at the Any data type and how it can be handled in the CORBA IDL to Java binding. We originally saw an IDL that used Any back in Listing 8.16. Let’s say, for example, that we want to send data of type “string”, then on the Client we could do:

```
Any mymsg=orb.create_any();  
mymsg.insert_string("hi  
there");
```

whereas if we wanted to send data of type “double”, then on the Client we would do:

```
Any othermsg=orb.create_any();  
othermsg.insert_double(3.1415);
```

Then on the servant if we *knew* the data we were getting was of type “string”, then we could do:

```
String themsg=mymsg.extract_string();
```

Whereas if we *knew* the data we were getting was of type “double”, then we could do:

```
double thefloat=othermsg.extract_double();
```

As before, I’m ignoring the situation where we *don’t* know what the data type is, and have to extract it from the TypeCode in the Any. That’s getting away from the static interfaces and more into the dynamic interfaces, and I’m largely ignoring those for this (introductory) textbook.

8.2.4 CORBA ADDRESSING—HOW DOES THE CLIENT FIND THE SERVER AND ASSOCIATED SERVANT?

The major address formats that a Client can use to find (address) a server/servant in CORBA are:

- Interoperable Object Reference (IOR) format:
 - Used with or without the original CORBA Naming Service
- Uniform Resource Identifier (URI) formats from the CORBA Interoperable Naming Service:
 - Corbaloc
 - Corbaname

The way an IOR, a corbaloc URI, or a corbaname URI are accessed on the Client is by using a CORBA function called `string_to_object`. That is, the appropriate address information is stored in a string, usually in one of the three formats above, then on the Client the string is converted into an object reference. Then this object reference is narrowed to a proxy object that points to the correct interface.

In Java, this looks like (here the variable “line” is of type `String`—also note the use of the `Helper` class to reference the interface):

```
org.omg.CORBA.Object obj = orb.string_to_object(line);
echo myecho = echoHelper.narrow(obj);
```

and in C++ this looks like:

```
CORBA::Object_var obj = orb->string_to_object (line.c_str());
echo_var myecho = echo::_narrow (obj);
```

In the C++ example, the variable “line” is a C++ string, and is converted to a C style string by the use of the “`c_str()`” function applied to the C++ string.

We will see the IOR, corbaloc, and corbaname formats and how they are handled below.

Other format options that can be used instead of IOR, corbaloc, or corbaname to be read by `string_to_object` are (these are optional and may be implemented by a vendor):

- `file://filename`
 - Name of a file that contains a stringified IOR or a URL of a CORBA object
- `http://url`
 - A URL that allows retrieval of a stringified IOR or a URL of a CORBA object
- `ftp://filename`
 - Can use ftp to retrieve a file that contains stringified IOR or a URL of a CORBA object

8.2.4.1 Interoperable Object Reference and the CORBA Naming Service

The Interoperable Object Reference (IOR) format is the only

way that CORBA had to begin with to pass address information around, starting back in 1991. For getting started, you can think simply of an IOR as containing the IP address and port number of the Server, together with an object reference that gives you a way, after you've connected to the IP address and port number, to find the correct servant object that implements the interface.

8.2.4.1.1 *The IOR Format*

Typically, the IOR is created on the Server, then “stringified” (that is, turned into a string of hex digits that begins with the characters “IOR:”) and then sent to the Client some way (via email for example, or via putting the stringified IOR in a file and transferring it with sftp).

An example of how to do this in C++ is:

```
ofstream thefile ("echo.ior");

CORBA::Object_var theref = mypoa->id_to_reference (object_identifier.in());
CORBA::String_var mystr = orb->object_to_string (theref.in());

thefile << mystr.in() << endl;
thefile.close ();
```

and how to do it in Java is:

```
FileWriter thefile = new FileWriter("echo.ior");
BufferedWriter bufferedWriter = new
BufferedWriter(thefile);

    org.omg.CORBA.Object theref = mypoa.id_to_reference(servant_id);
    String mystr = orb.object_to_string(theref);

bufferedWriter.write(mystr);
bufferedWriter.close();
```

In both cases, a file named “echo.ior” is created that contains the stringified IOR.

Also in both cases, we begin with an `object_identifier` that refers to the servant: it’s called “`object_identifier`” in the C++ code and “`servant_id`” in the Java code.

In both cases, we see that a CORBA routine called “`id_to_reference`” is used to translate the Object Identifier to an Object Reference (a pointer to a CORBA object, that is, something in CORBA that can contain pointers to a particular interface).

Finally, the object reference is “stringified,” or converted to a string, using the CORBA routine called “`object_to_string`” and written to the `echo.ior` file.

The `echo.ior` file is passed to the Client in some way, then the Client reads the file and reverses the process. We saw earlier in this section how a CORBA `string_to_object` routine is used in the Client to translate a string read from the `echo.ior` file into a proxy reference.

We have seen the most basic way in which an IOR is used. However, the IOR does more than just specify IP address port number, and object reference. For example, embedded inside the IOR are the `repository_id/type_id` that contains a unique identifier for the interface, and the version of the Internet Inter-Orb Protocol (IIOP) that is being used (I’ll talk

about repository_ids, which are related to Interface Repositories, and the IIOP later on in Section

8.2.7.1—note also that other transport protocols than IIOP could also be used), as well as the host address, port number, and object key (object identifier).

IORs are meant to be opaque, that is, the user isn't supposed to have to look at the internals of an IOR. However, there are utilities provided with various ORBs that allow you to view and interpret the contents of a “stringified” IOR. Some utilities include:

- cator—omniORB and the TAO ORB
- iordump—MICO and ORBacus
- There's an online web tool that will do it for you:
 - <http://www2.parc.com/istl/projects/ILU/parseIOR/>

(NOTE: omniORB also contains a utility called geniorb that will generate a stringified IOR if you give it a type_id, IP address, port number, and Object key. But you have to know what you're doing if you do that, plus the need to do that should seldom arise.)

One major additional capability of an IOR is that of “tagged profiles.” An IOR can contain several different sets of information on how to contact it, each stored in a tagged profile. For example, one tagged profile could use IIOP and a different tagged profile could specify the use of some other transport protocol. Or an IOR could contain two tagged profiles that use IIOP, but one of the profiles specifies some vendor-specific operation while the other tagged profile is a vanilla profile meant to connect with ORBs from other vendors. Probably, the most common use of tagged profiles would be for vendors to provide added features for customers who use their ORB on both Client and Server. I'm not telling you how to do that because you're a student and it's a little more than you need in an introductory class. One of those nasty little exercises for the reader things.

8.2.4.1.2 *The Original CORBA Naming Service*

Now, we will look at the CORBA Naming Service. The CORBA Naming Service works a lot like the Domain Name Service used in the World Wide Web: the Domain Name Service translates URLs (such as www.uah.edu) to their associated IP addresses, whereas the CORBA Naming Service is used to translate to/from names to object references. It's useful to use names rather than object references, because transferring IORs around can be a pain.

Note that I just snuck in a plug for my home university, UAH. Feel free to look up the Computer Science department here. ☺

If you're older folks like me, you remember the old White Pages and the Yellow Pages, provided by the telephone company. The old White Pages were the equivalent of the CORBA Naming Service; they map a person to address and phone number. If you don't remember what a White Pages was, don't worry about it. It's swiftly going over the event horizon.

(The old Yellow Pages had advertisements for products and services; they mapped service/product offerings to names/addresses/phone numbers. The CORBA Trading Service is an equivalent to the Yellow Pages. Although this is largely historical information, we will look at the Trading Service later on because it led to several spin-off technologies that are still in use in other places.)

The Naming Service runs as a separate executable. Each ORB vendor provides its own version of a CORBA compatible naming service.

Note that some ORB vendors may also provide proprietary (extra non-CORBA compliant) extensions to the Naming Service, and some ORB vendors may also provide their own proprietary naming service as a separate executable. Back in 2001, during a contract I had with the U.S. Army to determine

CORBA interoperability, I was trying to connect the Visibroker ORB to the TAO ORB. As I recall, all worked well as long as I was using the TAO Naming Service. I could use a Visibroker server to bind a Visibroker CORBA object (with associated servant) to a name on the TAO Naming Service, and alternately I could use a TAO server to bind a TAO CORBA object (with associated servant) to a name on the TAO Naming Service. However, when I swapped and tried to connect a TAO server to the Visibroker Naming Service, it just plain didn't work. After numerous hours, I finally found a footnote on about page 300 of the Visibroker manual that said I had to use a different executable program for the naming service if I wanted something CORBA compliant. Sigh.

When using the Naming Service, the Server provides a name and reference to a CORBA object to the Naming Service, such that later on, when a client calls the Naming Service and passes it that name, the Naming Service will return the associated object reference to the Client. When the Server does this, it is called *binding* the name to the object reference.

There is a call in the CORBA Naming Service interface called "bind" that will do this. However, if you call it again with the same name, it will return an `AlreadyBound` exception. Although I can see the utility of this in a production setting, this is a pain for a student who is developing code, because you may kill your server and start it again but not remember to kill the Naming Service. It's easiest to use the *other* Naming Service interface call, "rebind," because rebind won't fail on a second call using the same name, it just writes over the old reference. I highly recommend that students use "rebind" because I've seen students spend hours trying to figure out what's wrong when all they did was forget to restart the Naming Service.

However, although you can just bind a name to an object reference, it's not quite as simple as that overall. You can also create graphs of names in a hierarchy. This works sort of like a directory hierarchy. For example, in a directory hierarchy you

could have:

dirA/subdirA/myfile

and

dirA/subdirB/myfile

That is, since `subdirA` is different from `subdirB`, you can have a file named `myfile` under `subdirA` that is different from the `myfile` under `subdirB`.

The CORBA Naming Service works similarly in regard to hierarchy compared to how a directory structure works. Unfortunately, in CORBA we can't quite do it this way, since in the CORBA Naming Service they didn't reserve the slash marks `"/"` or `"\"` or for that matter any other character to serve as a separator between name components. So in CORBA, if your name has more than one component, you can't represent it in a single string. This is yet another thing CORBA did to try to be everything to everybody—the argument here was that you didn't know all the languages you were binding to, so you couldn't reserve a slash mark to mean anything special to CORBA. That is, a slash mark could be part of a valid name. For that matter, any kind of non-printing character can be part of a valid name in the CORBA naming service. Sigh.

The collection of names in a hierarchy is called a *naming graph*. A leaf node on this naming graph is a `Name`, whereas the base nodes and any intermediate nodes are called naming contexts. Using the similarity to a directory structure again, a naming context contains names (together with their associated bindings) and other sub-naming contexts like a directory would contain files and other subdirectories.

```

Base name contextA  → NameA
                    → sub name contextA → NameB
                                   → NameC
Base name contextB  → sub name contextB → NameB

```

Here, Base name context, Base name context, sub name context, sub name context are all name contexts that serve as starting points for reading a complete name (including the

hierarchy).

NameA, NameB, NameC, and NameD are names that are normally associated with object references, that have been bound to object references using either “bind” or “rebind.”

Sub name contextA contains two names, NameB and NameC.

Note that the sub-graph beginning with Base name contextB is not connected at all to the sub- graph beginning with Base name contextA. This is allowed. It’s reasonable since you might be run- ning several servers that are only loosely related, or potentially not related at all, so that their names don’t overlap.

A naming graph is allowed to have loops. Sigh.

The Naming Service interface is represented in an IDL. You can generally find this IDL file in your ORB source code. For example, the version of omniORB I’m currently using has a file called Naming.idl in the directory: /omniorb-4.2.1/idl. In the version of MICO I’m using, there is a file called CosNaming.idl in the directory /mico/include/coss.

The module name is CosNaming. A name in the CORBA Naming Service is stored in a struct called NameComponent:

```
typedef String lstring;
struct NameComponent {

    lstring id;

    lstring kind;

};
```

Then a complete Name (including the name hierarchy) is a sequence of NameComponents. The name of each separate NameComponent is stored in the “id” field.

The “kind” field is an annoying thing that apparently originally the authors of the CORBA specification meant to use to provide some description of what the “id” field was. I guess, for example, you could have an “id” field such as “Images” and a kind field that said “FileServer.” To the best of my knowledge, the “kind” field is not often used. It’s safest to always explicitly set it to the empty string.

LISTING 8.23 Bind an Object Reference to Simple Name

```
//Create a name for the CORBA object (and its servant)
    to be associated with on the Naming Service
CosNaming::Name
name; name.length
(1);
name[o].id = CORBA::string_dup (“echoservant”);
name[o].kind = CORBA::string_dup (“”); // make sure
```

In Listing 8.23, we see server code (using omniORB in C++) where an echo servant is bound to a simple name on the CORBA Naming Service. I’m calling it a “simple name” because it’s only using one base naming context (or initial naming context), and that doesn’t have a name (it’s an “orphaned” context) and one name. Because there’s only one name, we do:

```
name.length(1);
```

We’re assuming in Listing 8.23 that the variable “namingcontext” was determined before this code executes. We’ll look at how that works a little later. We’re also assuming that “echo_ref” is a CORBA object that was previously determined.

Note that I’ve explicitly set the “kind” field to empty string.

A string such as the “kind” field is supposed to default to

empty string (in CORBA 2.3 and later) but I always figure why take chances. I used “rebind” here. We talked about “bind” and “rebind” earlier, but those only bind the names (the leaf nodes on the naming graph) to the object references. For the name contexts, you use `bind_new_context`. Let’s suppose that I wanted the whole name to look like this:

`allechos → echoservant`

Then I would have done the code in Listing 8.24.

LISTING 8.24 Bind an Object Reference to Simple Name

```
// Create a name for the CORBA object (and its servant)
// to be associated with on the Naming Service
CosNaming::Name
name; name.length
(1);
name[0].id = CORBA::string_dup (“allechos”);
```

```

name.length (2);
name[1].id = CORBA::string_dup ("echoservant");
name[1].kind = CORBA::string_dup (""); // make sure
the "kind" field is empty

```

Note in Listing 8.24, I used “bind” instead of “rebind.” This is because I’m using `bind_new_context`. Since `bind_new_context` will return an `AlreadyBound` exception if the context name is already being used, I’ll already have that exception before I would get to the `rebind`, so no point in using a `rebind`.

Note that there are also separate routines to create a context (`new_context()`) and bind to an already existing context (`bind_context()`) if you wanted to do it that way instead.

Now, I said earlier that in Listing 8.23 the variable “`namingcontext`” was determined before the code executed. This can be done in a couple of different ways. The usually recommended one is to use a CORBA routine called “`resolve_initial_references.`” We see how this is done in Listing 8.24, this code was used on the C++ `omniORB`.

LISTING 8.25 Using Resolve Initial References to Find an Initial Naming Context

```

// Find the Naming
Service itself
naming_service
_ref =
    orb->resolve_initial_references ("NameService");

CosNaming::NamingContext_var
namingcontext=

```

In Listing 8.25, the call to `resolve_initial_references` is asking for the Naming Service.

(The `resolve_initial_references` call can also be used for other major CORBA functions such as “RootPOA.” We’ll discuss this later when we talk about CORBA server-side processing.) However, for `resolve_initial_references` to find the Naming Service, the Naming Service has to be configured somehow prior to this call. For example, one way the omniORB Naming Service, `omniNames`, works is that it reads a file called `omniORB.cfg` at startup. Inside this file an object reference to the root context of the Naming Service is available.

Sometimes there are ways that individual ORB vendors created to find the IOR of their Naming Service. For example, you will see in the code included with this textbook that `omniNames` can write out its initial object reference IOR when it is started, this can be saved to a file, and passed to Server and Client. Then the Server and Client can each be started using the command-line parameter:

```
-ORBInitRef NameService= ...object reference...
```

which will allow calls to `resolve_initial_references(“NameService”)`.

We'll see more how `-ORBInitRef` works in the next section, when we talk about the CORBA Interoperable Naming Service.

Similarly, the MICO Naming Service has a proprietary command-line parameter that can be used when calling the MICO Naming Service, that allows the initial object reference IOR to be written to a file:

```
nsd --ior Naming_Service.ref
```

where `nsd` is the MICO Naming Service, `"-ior"` is the proprietary command-line parameter, and `Naming_Service.ref` is the file name written to.

On the JDK Java ORB, the Naming Service (`orbd`) is started on a particular port number, using the following command-line parameter when the Naming Service is started, and again when the Client is started and when the Server is started:

```
-ORBInitialPort 1163
```

then as usual, the Client and Server each calls: `resolve_initial_references("NameService")`.

You can also use other methods specified in the Interoperable Naming Service to find the Naming Service, we'll talk about these in the next section when we talk about the Interoperable Naming Service.

The TAO ORB also supports another, different method for having a server or client find the TAO Naming Service, it uses an IP multicast. Basically, when a client or server calls `resolve_initial_references`, the TAO ORB sends out an IP multicast looking for any TAO Naming Service on the Local Area Network (LAN). (IP multicasting is a way to have one entity send IP packets addressed to several different recipients at the same time.) A running TAO Naming Service recognizes this IP multicast, and responds, with the result

that the `resolve_initial_reference` (“NameService”) that is called from the Server or Client connects automatically to the TAO Naming Service, without the need for other command-line parameters or passing any kinds of files around.

Back in the 2000 through 2004 or so timeframe, I often taught two classes at a time of 30+ students, who all used the TAO ORB at the same time, and who all started their own copies of the TAO Naming Service at similar times. It often occurred that one student’s code would connect to a second student’s instance of the TAO Naming Service, and the first student’s code would suddenly and (incomprehensibly to the student) stop working when the second student decided to shut things down and go to lunch. The TAO ORB allows an environment variable (called `NameServicePort`) to be set, to put the Naming Service on its own port and to allow Server and Client to look for the Naming Service on that port, but students almost never remembered to use this, no matter how often I reminded them. ☺ Of course there are numerous other ways for a student’s code to find that student’s Naming Service, but using the built in IP multicast always seemed so convenient to them. Until it suddenly wasn’t.

But let’s look back again now at Listing 8.24, at the call to `CosNaming::NamingContext::_narrow()` in the second statement. The `narrow()` routine does a type downcast from a general CORBA object to a derived object. In this case, the derived object is the `NamingContext`.

However, the `narrow()` routine does a whole lot more than a type downcast, at least when CORBA is using its more common synchronous communications (that is, a client calls a server and expects an immediate response) as opposed to asynchronous (the Client calls a Server but doesn’t expect an immediate response, and the Client goes on about its business). When using synchronous communications, the `narrow` results in data communications across the network (!).

The reason it does, the creators of CORBA decided that

CORBA had to check to make sure it was appropriate to do a type downcast before the type downcast occurred. Therefore, the

narrow calls across the communications link to make sure the object being called is actually there at the appropriate address, and that it has the appropriate type. So for the narrow to work, there has to be a server of some kind there running, and it has to be running something of the correct type!

The CORBA Specification, Version 3.3 Part 1: CORBA Interfaces says:

To do such down-casting in a type safe way, knowledge of the full inheritance hierarchy of the target interface may be required. The implementation of down-cast must either contact an interface repository or the target itself, to determine whether or not it is safe to down-cast the Client's object reference.

(If there's an asynchronous communication going on, the narrow() would return an un-type-checked stub.)

This confuses students terribly. I've had so many students tell me, I've traced my code to a narrow, which is supposed to do a type downcast, and the narrow is failing! How can a type downcast fail! The ORB itself must be broken, maybe you didn't compile it right.

The ones who delve a little further tell me, it's throwing a TRANSIENT exception (or sometimes depending on what they're trying to narrow, it's throwing an OBJECT_NOT_EXIST exception). Why does a type downcast throw a TRANSIENT (or OBJECT_NOT_EXIST) exception. The ORB must be broken!

What has happened in this particular case is that your Naming Service wasn't running (a TRANSIENT exception is thrown when the ORB tried to reach the Server and failed).

I already talked about narrow() briefly back in Section 8.2.2. We'll see this same discussion again later on, when we discuss how a client tries to narrow to find the reference for a servant: I'm going to repeat all this stuff about narrow there. In my experience, it's one of the more aggravating things that

students learning CORBA go through.

We've been talking so far about using individual Naming Services. That is, each Naming Service is a separate executable file, running independently of the Server and of the Client, but so far we've only been using one executing Naming Service. However, there are often reasons to use groups of Naming Services, with interconnected Naming Graphs: these are called *federated Naming Services*. Maybe you have a bunch of different departments in your company, and each runs and maintains its own Naming Service. So these may be interconnected to create a company-wide Naming Service. All you have to do is link the IORs of individual CORBA objects. CORBA doesn't care where your objects are located as long as the correct IORs are passed around (location transparency, again).

8.2.4.2 CORBA Interoperable Naming Service

(corbaloc and corbaname) The addressing formats corbaloc and corbaname are parts of the Interoperable Naming Service, which was added in the CORBA 2.4 specification back in 2000.

The corbaloc format looks generally either like this:

```
corbaloc:iiop: < host IP address>:< port number>/object_key
```

(without a port number specified, the port number defaults to port 2809) Or like this:

```
corbaloc:rir: < host IP address>:< port number>/object_key
```

whereas the corbaname format looks generally like this:

```
corbaname: <corbaloc format including object key as before> #name string
```

Some corbaloc examples using iiop are:

```
corbaloc:iiop:localhost:4321/theObjectKey
corbaloc::localhost:4321/theObjectKey
```

(if you leave off the iiop and do "::" instead it assumes iiop)

```
corbaloc:iiop:localhost/theObjectKey
```

(remember without a port number specified, the port number defaults to port 2809)

```
corbaloc:1.2@theserver.com:4321/theObjectKey
```

(this selects IIOP version 1.2. The default version of IIOP is version 1.0. We will discuss the Internet Inter-Orb Protocol (IIOP) later on in Section 8.2.7.1.)

You can also use corbaloc to specify an IPv6 address:

```
corbaloc:iiop:1.2@[ipv6 address]:port number/theObjectKey
```

As we discussed earlier, these can be accessed using the CORBA `string_to_object` routine. The `string_to_object` routine is smart enough to determine whether the string being passed to it is an IOR, a corbaloc, or a corbaname, and it handles the string differently, depending on which it is.

Let's look further at the corbaloc:rir format. Here, "rir" stands for "resolve initial references." There is another CORBA routine called "resolve_initial_references", we saw this back when we talked about the original CORBA naming service. If you have a corbaloc URL that uses "rir," and you pass this corbaloc URL to the CORBA `string_to_object` routine, then the `string_to_object` routine will first pass the corbaloc URL to `resolve_initial_references`, and create the object reference using the returned value. For example, it would be reasonable to do: `corbaloc:rir:/NameService` to find the CORBA Naming Service.

However, normally you would use corbaname to find a name on the CORBA Naming Service.
Here are two examples:

```
corbaname::theServer.edu:1234#echo
```

Here, “echo” is the name of the CORBA object you’re looking for on the Naming Service, whereas “theServer.edu:1234” is the URL including port number of the Naming Service itself. Note that we’re doing a “:.” here so it’s defaulting to “iiop”. It’s also defaulting to port number 2809.

`corbaname::localhost:1234#echo`

Says that

your naming service is running on localhost at port 1234 and you’re looking for the server/servant named “echo”.

You could also do:

`corbaname:rir: /NameService#echo`

If you use the corbaname “rir” format, then what it means is the Naming Service is found using: `resolve_initial_reference(“NameService”)` prior to passing “echo” to the Naming Service. (So you can actually leave off the “/NameService” here, it’s the default.)

The Interoperable Naming Service specification also allows the location of particular services to be set using command-line parameters to the ORB. For example, you can do the following:

- `-ORBInitRef`
`NameService=IOR:... or`
- `-ORBInitRef NameService=corbaloc::myserver.edu/NamingService`

After you've done this, however, then `resolve_initial_references("NameService")` will return the `NamingService` previously specified using `-ORBInitRef`. This can be done for any CORBA service, not just the Naming Service.

LISTING 8.26 Using `corbaloc` to Find an Initial Naming Context

```
org.omg.CORBA.Object naming_service_ref =
    orb.string_to_object("corbaloc:iiop:1.2@localhost:1163/NameService");
```

In Listing 8.26, we see how `corbaloc` can be used with the JDK Java ORB to find the orbd Naming Service. This assumes that orbd is running at port 1163, and that JDK is using IIOP version 1.2 to find the Naming Service. (Actually it's sending a `locateRequest` with `parameter = NameService`, we'll look at `locateRequest` later on when we look at GIOP in Section 8.2.7.1.)

This assumes you previously started orbd on port 1163 as follows:

```
orbd -ORBInitialPort 1163
```

The `narrow()` works as before in Listing 8.24, see Section 8.2.4.1.2.

LISTING 8.27 Using `corbaname` to Find an Initial Naming Context

```
org.omg.CORBA.Object proxy =
```

In Listing 8.27, we see how `corbaname` can be used with the JDK Java ORB to find the orbd Naming Service. The portion `"corbaname::localhost:1163"` says that the naming service is located at localhost on port 1163. The text `"echo"` after the `"#"`

specifies the name of the CORBA object (with associated servant) that was previously attached to the naming service by the server.

Again, the `narrow()` works as before in Listing 8.26 and Listing 8.24.

8.2.5 SIMPLEST ECHO EXAMPLE

We're now going to look again at the Client, Server, and servant code that goes with the simplest echo interface previously shown in Listing 8.1. Since you've read everything between when the simplest echo interface originally showed up back in Section 8.2.1.1, you now know enough to understand this. ☺ We're going to look at both the C++ and Java versions, in two C++ ORBS (MICO and omniORB) and one Java ORB (JDK ORB).

8.2.5.1 Echo Example—Server Side

We're going to start with the Server/Servant code. In Listing 8.28, we see the Server/Servant code for the C++ MICO ORB. In Listing 8.29, we see the Server/Servant code for the C++ omniORB. I previously showed you the Java Server and Servant code for the JDK ORB back in Listing 8.2 and Listing 8.5 in Section 8.2.1.1, but I'm going to show it in a listing again here, so you can easily

compare it to the C++ versions (plus the way the code is written, it really belongs in a single listing instead of two).

In the Server, the first thing that happens is initializing the ORB. This is done in C++ by a call to:

```
CORBA::ORB_var orb CORBA::ORB_init (argc, argv);
```

And in Java by a call to:

```
ORB orb = ORB.init(args, null);
```

Both these examples allow command-line parameters to be used to initialize the ORB. For example, on the JDK Java ORB, assuming the Naming Service (orbd) is started on a particular port number (1163), when you pass in the following command-line parameter as an argument, this initial port information will be provided to the ORB initialization routine in the “args” parameter:

```
-ORBInitialPort 1163
```

Note that we could alternately have done something like (in this case, in Java, to do this you would also have to import `java.util.Properties`):

```
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialPort", "1163");
ORB orb = ORB.init(args, props);
```

Here the initial port information is provided to the ORB initialization routine in the “props” parameter.

Note that in both C++ and Java, the ORB is just another object, it contains data and has associated routines.

The next thing to do on the Server is to get a pointer to the Root POA and to the POA Manager.

In C++ this looks like:

```
CORBA::Object_var the_rootPOA = orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var mypoa = PortableServer::POA::_narrow (the_rootPOA);
PortableServer::POAManager_var mgr = mypoa->the_POAManager();
```

and in Java this looks like:

```
org.omg.CORBA.Object the_rootPOA=orb.resolve_initial_references("RootPOA");
POA mypoa = POAHelper.narrow(the_rootPOA);
mypoa.the_POAManager().activate();
```

We talked earlier about how the POA does a lot of server-side functions, the most basic of which is mapping the interface to a CORBA object and from there to a Servant. It does a lot more than that, we'll discuss it in much more depth in Section 8.2.8. But before you can do anything at all involving a POA, you have to create a Root POA object and activate it. You can have other POA objects that are children of the Root POA, but the Root POA is the most basic POA and must be defined first.

The way to find a root POA is through `resolve_initial_references("RootPOA")`. Note that `resolve_initial_references("RootPOA")` works differently from how `resolve_initial_references("NameService")` works, in that with the RootPOA, everything is built into the ORB, whereas the

Naming Service runs as a separate executable. So you *don't* have to start a root POA running before you call `resolve_initial_reference("RootPOA")`—just go ahead and do the call.

The narrow routine does a type downcast as before. However, since the RootPOA (and for that matter all POAs) are built into CORBA, in this case the narrow doesn't have to do any across the communication link checks—so you don't have to worry about this call to narrow throwing a TRANSIENT exception. ☺

In the last step, the POA Manager is created and activated, and the POA Manager activates the RootPOA (makes the RootPOA start working). The POA Manager controls one or more POA objects. The POA Manager controls whether requests to a POA are queued for the POA or discarded. The POA Manager activates the POA and deactivates the POA.

As we said before, you can have other POAs that are children of the Root POA. The reason you might want to have multiple POAs is that you may want different groups of servants to be treated in a different way, to have different behaviors as to how they start up, what their life-time is, etc. These different behaviors would be set as policies on the POA that controlled the groups of servants. (We will examine how these policies work in Section 8.2.8.) The idea is that one POA would be created with a set of policies, and the servants that obey those policies are controlled by that POA. A different POA would be created with a different set of policies, and the servants that obey those policies are controlled by this POA.

POAs are set with certain policies by default, when they are created. For example, one default policy of a POA would be `SYSTEM_ID` for the Id Assignment policy. This allows CORBA to automatically define a name for a CORBA object rather than requiring the programmer to do so. This is useful for transient objects that only last the lifetime of the Server. We're

doing that in this case: the default value of the LifeSpan policy is TRANSIENT, so since we didn't tell the POA otherwise, we're using a Lifespan Policy of transient, and so our object will last only as long as the Server lasts.

The child POAs do *not* inherit the policies of the parent POAs. Instead, they are initialized with the default POA policies, unless certain policies are explicitly set when the child POA is created.

In this example, after we activate the POA, we define the servant (the_echo), and activate it using explicit activation:

```
PortableServer::ObjectId_var object_identifier = mypoa->activate_object (the_echo);
```

This tells the POA that when a request comes in for a CORBA object, it's going to be forwarded on to the servant the_echo.

Then we save the IOR of the activated CORBA object to a file so the Client can eventually find it.

Finally, we set the POA looking for input requests from a Client.

LISTING 8.28 Echo Example, Server and Servant, MICO ORB, in C++/C

```
#include  
"echo.h"  
#include  
<fstream>  
using  
namespace std;
```

```

char * myecho::echostring (const char *the_echostring)
{
    cout << "string received is " <<
    the_echostring << endl; char
    *smsg=
    CORBA::string_dup(the_echostring); return smsg;
}
// This is the Server
int main (int argc, char *argv[])
{
    // Begin by initializing the ORB
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get POA and the POA Manager
    CORBA::Object_var the_rootPOA = orb-
    >resolve_initial_references ("RootPOA");
    PortableServer::POA_var mypoa =
    PortableServer::POA::_narrow (the_rootPOA);
    PortableServer::POAManager_var mgr = mypoa-
    >the_POAManager();

    // Create a servant
    myecho * the_echo = new myecho;
    // use explicit activation to map the CORBA object to
    the servant PortableServer::ObjectId_var
    object_identifier = mypoa->activate_object
    (the_echo);

    // Save the IOR in a

```

```

file ofstream
thefile
("echo.ior");
CORBA::Object_var theref = mypoa-
>id_to_reference (object_identifier.in());
CORBA::String_var mystr = orb-
>object_to_string (theref.in());
thefile <<
mystr.in() <<
endl; thefile.close
();

// Start the POA, start
looking for input requests
mgr->activate ();
orb->run();

mypoa->destroy
(TRUE, TRUE); delete
the_echo;

return 0;
}

```

LISTING 8.29 Echo Example, Server and Servant, omniORB, in C++/C

```

#include
"echo.hh"
#include
<iostream>
#include
<fstream>
using
namespace
std;

#define TRUE 1
#define FALSE 0
// This is the Servant
class myecho : public virtual POA_echo
{
public:
    virtual char * echostring (const char *the_echostring);
};
char * myecho::echostring (const char *the_echostring)
{
    cout << "string received is" <<
    the_echostring << endl; char
    *smsg=
    CORBA::string_dup(the_echostring); return smsg;
}
// This is the Server
int main (int argc, char *argv[])
{
    // Begin by initializing the ORB
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

    // Get POA and the POA Manager
    CORBA::Object_var the_rootPOA = orb-
    >resolve_initial_references ("RootPOA");
    PortableServer::POA_var mypoa =
    PortableServer::POA::_narrow (the_rootPOA);
    PortableServer::POAManager_var mgr = mypoa-
    >the_POAManager();

```



```

// Create a servant
myecho * the_echo = new myecho;

// use explicit activation to map the CORBA object to
the servant PortableServer::ObjectId_var
object_identifier = mypoa->activate_object
(the_echo);

// Save the IOR in a
file ofstream
thefile
("echo.ior");
CORBA::Object_var theref = mypoa-
>id_to_reference (object_identifier.in());
CORBA::String_var mystr = orb-
>object_to_string (theref.in());
thefile <<
mystr.in() <<
endl; thefile.close
();

// Start the POA, start
looking for input requests
mgr->activate ();
orb->run();

mypoa->destroy
(TRUE, TRUE); delete
the_echo;

return 0;
}

```

LISTING 8.30 Echo Example, Server and Servant, JDK ORB, in Java

```
import
org.omg.CORBA.*;
import
org.omg.PortableSer
ver.*;
import org.omg.PortableServer.POA;

import
java.util.Properties;
import java.io.*;

// This is the Servant
class myecho extends echoPOA {
    public String echostring(String
        the_echostring) {
        System.out.println(the_echo
            string);
        return the_echostring;
    }
}

// This is the
Server
public class
server {
    public static void

    main(String args[]) { try

    {

        // Begin by
        initializing the ORB
        ORB orb =
        ORB.init(args,
```

```

null);
// Get POA and the POA Manager
    org.omg.CORBA.Object
        the_rootPOA=orb.resolve_initial_references
            ("RootPOA");
    POA mypoa =
        POAHelper.narrow(the_rootPOA);
    mypoa.the_POAManager().activate();
                                                //activate the
        POA manager
// create servant and register
    it with the ORB
    myecho
    the_echo=new
    myecho();

// get object reference from the servant
    byte [] servant_id =
        mypoa.activate_object(the_echo);

// Write the IOR to a file
    String line=null;
    FileWriter thefile = new
    FileWriter("echo.ior"); BufferedWriter
    bufferedWriter = new
    BufferedWriter(thefile);
        // since only writing one value probably not
        // necessary but I'm used to
        // using bufferedWriter around all FileWriters
    org.omg.CORBA.Object theref =
        mypoa.id_to_reference(servant_id);
// convert the servant ID to a
    CORBA object reference String
    mystr =
        orb.object_to_string(theref);
// Convert the CORBA object reference to a string
    bufferedWriter.write(mystr);

    bufferedWriter.close();

    System.out.println("Server is
    waiting for input");

```

```

        orb.run();
    }
    catch (Exception the_exception)
    {
    }
}

```

8.2.5.2 Echo Example—Client Side

Here we see the Clients that go with the Server/Servant for each of the three ORBs: MICO ORB, omniORB, and JDK ORB. Refer to Listing 8.31 for the MICO ORB, Listing 8.32 for omniORB, and Listing 8.33 for the JDK ORB.

If you want to run these, you *DON'T* have to use the same language and ORB for both Client and Server. For example, try connecting the JDK ORB to omniORB. It will work. ☺

Remember that you

have to move the IOR file from the directory where the Server created it, to the directory where the Client will read it.

In all three examples, we do the following steps:

- Initialize the ORB
- Retrieve the stringified IOR from the file where the server originally wrote it
- Convert the IOR into an object reference
- Using `narrow()`, type downcast the object reference to be an object of the type of the echo interface. (Remember that the `narrow()` routine will do a call across the communications link to check the type, and if the server isn't running this call to `narrow()` will throw an exception.)
- Do a remote procedure call to the `echostring` routine on the Servant

Note that there's not a lot of difference between the C++ versions and the Java version. Do note that you have to use `echoHelper` to call the `narrow()` routine when you're using Java and the JDK ORB.

LISTING 8.31 Echo Example, Client, MICO ORB, in C++/C

```
#include
"echo.h"
#include
<unistd.h>
#include
<fstream>
using
namespace
std;

int main (int argc, char *argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init (argc,
```

```

if (CORBA::is_nil (myecho)) {

    cout << "the server is
    misssing" << endl; exit
    (1);
}
CORBA::String_var the_echostring = myecho-

```

LISTING 8.32 Echo Example, Client, omniORB, in C++/C

```

#include
"echo.hh"
#include
<stdio.h>
#include
<stdlib.h>
#include
<iostream>
#include
<unistd.h>
#include
<fstream>
using
namespace
std;

int main (int argc, char *argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init (argc,
    argv);

    // Get the IOR
    from a file

```

Let's revisit the narrow() routine again. I'm going to repeat

(with a few small changes and expansions) a little discussion that I previously gave when I described narrow()-ing to a Naming Service. The reason I'm going to repeat it here is, using a narrow() on the Naming Service, and using a narrow() trying to call the Server direct (with an IOR from a file) are two instances of this issue where I've seen students become very confused. I want to make sure you've read it.

LISTING 8.33 Echo Example, Server and Servant, JDK ORB, in Java

```
import
org.omg.CORBA.*;
import java.io.*;

public class client

{
    public static void main(String args[])
    {
        try
        {
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Get the IOR
            from a file
            String
            line=null;
            FileReader thefile = new FileReader("echo.ior");
            BufferedReader bufferedReader = new
            BufferedReader(thefile);

            line=bufferedReader
            .readLine(); if
            (line==null)
                System.out.println("no IOR found in IOR
            file").
```

As you recall if you've previously read the section on the Naming Service ☺, the narrow() routine does a whole lot more than a type downcast, at least when CORBA is using its more common synchronous communications (that is, a client calls a server and expects an immediate response) as opposed

to asynchronous (the Client calls a server but doesn't expect an immediate response, and the Client goes on about its business). (When you're doing asynchronous communication, you can use the unchecked `narrow()`, which simply does a type downcast. The actual type check then wouldn't occur until the remote procedure is actually called on the object.) When using synchronous communications, the `narrow` results in data communications across the network (!).

The reason it does, the creators of CORBA decided that CORBA had to check to make sure it was appropriate to do a type downcast before the type downcast occurred. Therefore, the narrow calls across the communications link to make sure the object being called is actually there at the appropriate address, and that it has the appropriate type. So for the narrow to work, there has to be a server of some kind there running, and it has to be running something of the correct type!

The CORBA Specification, Version 3.3 Part 1: CORBA Interfaces says:

To do such down-casting in a type safe way, knowledge of the full inheritance hierarchy of the target interface may be required. The implementation of down-cast must either contact an interface repository or the target itself, to determine whether or not it is safe to down-cast the client's object reference.

This confuses students terribly. I've had so many students tell me, "I've traced my code to a narrow, which is supposed to do a type downcast, and the narrow is failing! How can a type downcast fail!" The ORB itself must be broken, maybe you didn't compile it right.

The ones who delve a little further tell me, it's throwing a TRANSIENT exception (or sometimes depending on what they're trying to narrow, it's throwing an OBJECT_NOT_EXIST exception). Why does a type downcast throw a TRANSIENT (or OBJECT_NOT_EXIST) exception. The ORB must be broken!

What has happened in this particular case is that your Server wasn't running (a TRANSIENT exception is thrown when the ORB tried to reach the Server and failed).

8.2.6 ECHO EXAMPLE USING NAMING SERVICE

In this section, we will see a variation on the echo example, but this time instead of passing around an IOR, we're going to use a Naming Service. We're only going to look at the version in Java

using the JDK ORB here. Back in Section 8.2.4.1.2 when we originally discussed the Naming Service, all the examples were in C++. Also, this is a complete example of Server/Servant and Client using the Naming Service, whereas in Section 8.2.4.1.2 we were looking at code fragments only. Be sure to compare how the Naming Service is accessed and used in Java to how the Naming Service is accessed and used in C++.

Here in the Server/Servant code shown in Listing 8.34, the Servant is called “myecho” and extends echoPOA. The echoPOA was automatically generated by the IDL compiler. The Server itself does the following steps:

- Initializes the ORB
- Sets properties for the POA so that it knows where to find the Naming Service (IP address and port number)
 - If it's done this way, then the Server can be started without using the command-line parameters: `-ORBInitialPort 1163 -ORBInitialHost localhost` (compare this example to the example given in the code provided for the instructor with this textbook).
- Gets pointers to the Root POA and the POA Manager, and starts the POA Manager (this is the same as in Section 8.2.5.1, simplest echo example). We'll see much more about the POA later in Section 8.2.8.
- Creates a servant object (the_echo)
- Registers the CORBA object associated with the servant on the POA (set it ready to receive input from the client)
- Uses `resolve_initial_references` to find the Naming Service, create an `object_reference` pointing to it
- Creates a “name” (NameComponent) with `id = “echo”` and `kind = empty string`
- Uses a “rebind” to map the NameComponent to the Servant CORBA object, on the Naming Service
- Starts the Server listening for requests from the Client

LISTING 8.34 Echo Example, Server and Servant, JDK ORB, in Java

```
import org.omg.CosNaming.*;
import
org.omg.CosNaming.NamingContextP
ackage.*; import org.omg.CORBA.*;
import
org.omg.PortableServer.
*; import
org.omg.PortableServer.
POA;

import
java.util.Properties; import java.io.*;

class myecho extends echoPOA {
    public String echostring(String the_echostring) {

        System.out.println(the_
echostring); return
the_echostring;
    }
}

public class server {

    public static void
        main(String args[]) {
        try {
        // Begin by initializing the ORB

        // These properties let the server find the orbd naming
        service
```

```

// (later on in the code) using
    resolve_initial_references
    Properties props =
    System.getProperties();
    props.put("org.omg.CORBA.ORBInitialPort", "1163");
    props.put("org.omg.CORBA.ORBInitialHost", "localhost"); ORB orb =
    ORB.init(args, props);

// Get POA and the POA Manager
    org.omg.CORBA.Object
        the_rootPOA=orb.resolve_initial_references
        ("RootPOA");
    POA mypoa =
    POAHelper.narrow(the_rootPOA);
    mypoa.the_POAManager().activate();;
                                                //activate the
    POA manager

// create servant
    myecho the_echo=new myecho();

// get object reference from the servant
    and register it on the POA byte []
    servant_id =
    mypoa.activate_object(the_echo);

    org.omg.CORBA.Object theref =
    mypoa.id_to_reference(servant_id);
        // convert the servant ID to a
    CORBA object reference echo echo_ref =
    echoHelper.narrow(theref);

```

```
// Find the Naming Service, it's part of orbd
    org.omg.CORBA.Object naming_service_ref =
        orb.resolve_initial_references ("NameService");

    NamingContextExt namingcontext=
        NamingContextExtHelper.narrow
            (naming_service_ref);

// bind the object reference to the naming service
    NameComponent name = new
        NameComponent("echo", "");
        // set name field to "echo" and kind
        field to empty string NameComponent path[] =
            {name};
        // Put name in an array of names,
        needed to call the rebind
        namingcontext.rebind(path, echo_ref);

    System.out.println("Server is
```

Here in the client code shown in Listing 8.35, the Client does the following steps:

- Initialize the ORB
- Use `resolve_initial_references` to find the Naming Service, create an `object_reference` pointing to it
- Create a "name" (`NameComponent`) with `id = "echo"` and `kind = empty string`
- Resolve the echo name on the Naming Service (use an `echoHelper` to do this) to create a proxy object called `myecho`
- Call the remote procedure `echostring`, using the proxy object `myecho`

LISTING 8.35 Echo Example, Client JDK ORB, in Java

```
import org.omg.CosNaming.*;  
import  
org.omg.CosNaming.NamingContext  
Package.*; import  
org.omg.CORBA.*;  
  
import
```

```

    try
    {

// create and initialize the ORB
        ORB orb = ORB.init(args, null);

// Find the Naming Service, it's part of
        orbd org.omg.CORBA.Object
        naming_service_ref =
            orb.resolve_initial_references("NameService
            ");

        NamingContextExt namingcontext=
            NamingContextExtHelper.narrow
            (naming_service_ref);
// Create a name to be used for searching the
        Naming Service NameComponent
        name = new
        NameComponent("echo","");
            // set name field to "echo" and
            // kind field to empty string
        NameComponent path[] = {name};
            // Put name in an array of names
    }
}

```

8.2.7 CORBA UNDER THE HOOD AND SOME LEFTOVER CORBA STUFF

We're going to look briefly here at the underpinnings of CORBA, it's a lot for beginners but it's important to know this because later on we'll have to use it to compare different middlewares. Then we'll look at a few kind of leftover topics that I couldn't figure out a good place to put otherwise. ☺

8.2.7.1 CORBA Under the Hood

General Inter-ORB Protocol (GIOP) is what CORBA uses instead of HTTP. It's got some in common with HTTP, in that it runs on top of TCP/IP (by default, anyway) and sends messages between a client to a server. However, the messages are fairly

different from HTTP messages, and GIOP is not associated with web pages.

GIOP consists of eight GIOP messages to manage communication between Client and Server and a CDR (a data representation for data being sent over a communications link).

The GIOP messages are the following:

1. Request—from Client to Server, used to call a remote procedure on an object on the Server
 - a. Includes a request_id, this is used to associate a later Reply with this Request
 - b. NOTE: later versions of GIOP allow a servant to send a Request on a connection that a client previously opened.

2. Reply—from Server to Client, contains the response to the remote procedure call:
 - a. Return value
 - b. Out parameters
 - c. Inout parameters
 - d. Exceptions if they occur
 - e. Includes a request_id, this is used to associate the Reply with a previous Request
3. CancelRequest—from Client to Server, tells the Server that the Client no longer expects to receive a Reply to a previous Request or LocateRequest
 - a. Includes a request_id, this is used to associate the CancelRequest with a previous Request
4. LocateRequest—from a client to a server, to ask the Server whether it could handle a Request to a particular object reference. The Server replies with a LocateReply.
 - a. Includes a request_id, this is used to associate a later Reply with this Request
5. LocateReply—from a server to a client, in response to a LocateRequest. The Server replies with one of:
 - a. UNKNOWN_OBJECT—the Server doesn't recognize that object reference
 - b. OBJECT_HERE—the Server is able to receive requests for that object reference
 - c. OBJECT_FORWARD—includes the IOR of a server where that object reference is located
 - d. Includes a request_id, this is used to associate the LocateReply with a previous LocateRequest
6. CloseConnection—from Server to Client, tells the Client that the Server is no longer processing requests, and no more replies will be forthcoming
7. MessageError—bad GIOP message
8. Fragment—follows a previous Request or Reply message, allows the Request or Reply to be broken into multiple chunks called Fragments. This fixes a problem where the Request or Reply had to know up front at the time the message header is being created, how many bytes (octets) would be in the message.

There is a general GIOP message header that is sent at the beginning of each of these messages. Among other items, the message header includes a field called (in later versions of GIOP) “flags.” The next to least significant bit in this field says whether any more fragment messages follow (0=no more fragments, 1=more fragments). The least significant bit of this field specifies whether the data in the message is in big-endian format or little-endian format.

Common Data Representation (CDR) is a format that is used to store IDL data when the data is in transmission over the

communication link between the Client and the Server and vice versa. Since the GIOP message header “flags” field tells the receiver whether or not the data in the message is being sent is big-endian or little-endian (see Sidebar on Big-endian and Little-endian in Chapter 1). Then if both sides are little-endian, no conversion of data is necessary, and if both sides are big-endian, no conversion of data is necessary. If one side is big-endian and the other side is little-endian, then the receiver is responsible for converting the data in the message to the appropriate endianness, according to the local format.

CDR specifies the number of bytes (octets) in the data stream allocated to each primitive IDL type. If the data isn’t long enough for its specified data type, the data in the data stream is padded, so that the primitive data types are always on the appropriate boundaries. For example, an IDL short is 2 bytes, a long is 4 bytes, a float is 4 bytes, and a double is 8 bytes.

The elements of a Struct are sent in CDR in the order in which they are defined within the Struct. Each individual element within the Struct is sent in the appropriate order according to its own type. Array elements are sent in sequence, with each individual element sent in the appropriate order according to its own type. The rightmost index varies the quickest, the leftmost index varies the slowest.

Sequences are sent as followed:

- Unsigned long, 4-byte length field, specifies number of elements in the sequence
- Elements of the sequence, with each individual element sent in the appropriate order according to its own type

IDL strings are sent in CDR as follows:

- Unsigned long, 4-byte length field, specifies length of the string
- Bytes of the string
- Null byte (note that the string length field counts this byte)

Internet Inter-ORB Protocol (IIOP) maps GIOP onto TCP/IP (specifies how TCP/IP connections are used to transmit GIOP messages). This defines the format of IORs for use in GIOP over TCP/IP. We previously discussed IORs in Section 8.2.4.1.

8.2.7.2 Some Leftover CORBA Stuff

The *Interface Repository* is used to contain all IDL type definitions so it can be queried for which interfaces are available and what they look like. The Interface Repository is itself a CORBA server and obeys the CORBA Interface Repository Interfaces.

When an IDL file is compiled using the IDL compiler, its type definitions can be stored in an Interface Repository to be retrieved by other Clients or Servers. The IDL compilers create unique identifiers for each IDL type. For example, each interface in an IDL file would have its own repository id: the omniORB IDL compiler for the interface shown in Listing 8.1 generates the following repository id (in the echoSK.cc file) and the JDK ORB IDL compiler also generates the same repository id (in the echoPOA.java file):

```
IDL:echo:1.0
```

The *Dynamic Invocation Interface* (DII) is the client-side interface for dynamic applications that discover interfaces at runtime. That is, using DII, the Client doesn't have to know

what the interface looks like at compile time, but can read the interface description sometime later on.

The base CORBA::Object implements the DII, so the calls to create a request object, and look at the return type of the request object, are invoked on an instance of CORBA::Object.

Let's look at Listing 8.36, which represents a client-side call using the interface defined in Listing

8.1. The DII calls on the CORBA::Object here are:

- `_request`(name of interface to call)—creates an empty request object
- `add_in_arg`()—adds an *in* argument to the request (creates an empty argument, returns

its data type as “Any&”, then the Any insertion operators write a string value into this argument)

- Arguments must be added in the order they appear in the IDL interface
- Other options would be:
 - `add_out_arg`()
 - `add_inout_arg`()
- `set_return_type`(CORBA::TypeCode)—initialize the return type on the request object
- `invoke`()—perform the remote procedure call
- `return_value`()—once you're sure there was a normal return (no exceptions), extract the
return value from the Any

There are also calls to the return environment to check for exceptions:

- `CORBA::Environment_ptr the_env = myrequest->env ();`

`CORBA::Object_var myobj=...` find object reference, perhaps using Naming Service as before

LISTING 8.36 Dynamic Invocation Interface Example, in C++

```
// Create the Request
CORBA::Request_var
myrequest=myobj-
>_request("echostring"); myrequest-
>add_in_arg() <=<= "hi there";
myrequest->set_return_type(CORBA::_tc_string);

// Perform the remote
procedure call
myrequest->invoke();

// Check for exceptions
CORBA::Environment_ptr the_env = myrequest->env ();
// check the returned
environment
```

The *Dynamic Skeleton Interface* (DSI) is the server-side interface for dynamic applications that implement interfaces which they discover at runtime. We will not look in depth at this interface, this is getting fairly advanced for a beginner textbook. However, just a short overview to get you started if you ever need to work on this (code snippets here are in C++).

The DSI servant must inherit from `PortableServer::DynamicImplementations`.

The `CORBA::ServerRequest` class is what is used to define a

Request object once the request has reached the server-side. When the POA receives a request, it will pass an instance of `ServerRequest` to the servant, using:

- `invoke (ServerRequest_ptr server_request);`

After the servant has figured out which operation it must implement, it has to retrieve the arguments for that operation. So the servant tells the ORB the number of arguments, their type, whether they're in, inout, or out using `ServerRequest::arguments`.

Inside the `invoke` routine which is implemented in the servant, to create the description of the arguments in order to query the ORB, do:

- `CORBA::NVList_ptr args;`
- `orb->create_list(o, args);`

The purpose of the `CORBA::NVList` object is to describe arguments in the DSI and DII.

Then create an Any data value, and do something like (this code assumes we're doing an echostring as in Listing 8.1):

- `the_any.replace(CORBA::_tc_string,0);`
- `args->add_value("", the_any, CORBA::ARG_IN);`

and finally call, to extract the argument values from the ORB:

- `server_request->arguments(args);`

Then to extract the values from the arguments, do something like:

- `args->item(o)->value()`

Then do the required processing to implement the request. After the required processing is complete, create a pointer to another Any data type:

- `CORBA::Any *the_result_any;`

Put the return data in the `_result_any`, and then call:

- `server_request->set_result(*the_result_any);`

8.2.8 PORTABLE OBJECT ADAPTER

As we discussed earlier, the Portable Object Adapter (POA) is concerned with how Servers and Servants handle the requests from the Clients. At least one POA per Server (the Root POA) is required.

The policies of the POA are concerned:

- With how servants are mapped to the CORBA objects that represent the interface (Implicit Activation Policy, Servant Retention Policy, Request Processing Policy)
- With the lifespan of the CORBA object that represents the interface, and the associated servant (Lifespan policy)

- With whether a CORBA object and servant can share the same identifier or must have unique identifiers (Id Uniqueness policy), and whether the system or the user assigns the identifiers (Id Assignment policy)
- With whether the server-side CORBA ORB can handle multiple requests at the same time, or whether all requests must be queued (Thread Policy)

Note that several of these policies are interacting. For example, which values you chose for the Id Uniqueness and Id Assignment policies are heavily related to which values you chose for the Lifespan policy. We'll discuss these in the following sections.

Also in the following sections, we'll be looking at *why* and *how* the POA does certain things certain ways. That is, we'll be learning how the POA in CORBA operates. It's convenient to do that as part of a discussion about the POA policies, because the policies control the behavior of the POA.

8.2.8.1 Introduction to POA Policies

All POAs, by default, are set to the following policies: TRANSIENT, USE_ACTIVE_OBJECT_MAP_ONLY, SYSTEM_ID, UNIQUE_ID, RETAIN, and ORB_CTRL_MODEL.

The Root POA, by default, is set to the IMPLICIT_ACTIVATION policy, whereas all other POAs are set to NO_IMPLICIT_ACTIVATION by default.

8.2.8.1.1 *Servant Retention Policy*

Possible values are RETAIN and NON_RETAIN. The default value is RETAIN. With RETAIN, the POA maintains an Active Object Map. With NON_RETAIN, no Active Object Map is used.

In CORBA, “activation” means that a servant is associated with (maps to) a CORBA object that represents the interface. The CORBA terminology is that the servant *incarnates* the CORBA object. We’ll talk more about how this works later on.

An Active Object Map is stored inside the POA, and maps CORBA objects that represent interfaces to the servants that incarnate them. When an incoming request related to a particular interface is received from the Client, the Active Object Map is used to map this request to the appropriate servant. If the servant is not found, an exception is thrown.

8.2.8.1.2 *Request Processing Policy*

Possible values are USE_ACTIVE_OBJECT_MAP_ONLY, USE_DEFAULT_SERVANT, and USE_SERVANT_MANAGER. The default is USE_ACTIVE_OBJECT_MAP_ONLY.

If USE_ACTIVE_OBJECT_MAP_ONLY is selected then the Servant Retention Policy must be set to RETAIN. Normally the id uniqueness policy will be set to UNIQUE_ID.

If USE_DEFAULT_SERVANT is selected, then:

- If the Servant Retention policy is set to RETAIN, then a request from the Client related to a particular interface is looked up first in the Active Object Map. If the CORBA object that implements that interface is not found in the Active Object Map, or if the CORBA object is in the Active Object Map but has not been incarnated by a servant, then the request is sent to the specified default servant.
- If the Servant Retention policy is set to NON_RETAIN, then a request from any client is sent to the specified default servant.

If USE_SERVANT_MANAGER is selected, then:

- If the Servant Retention policy is set to RETAIN, then a request from the Client related to a particular interface is looked up first in the Active Object Map. If the CORBA object that implements that interface is not found in the Active Object Map, then the servant manager is called. In this case, the servant manager supports the Servant Activator interface.
- If the Servant Retention policy is set to NON_RETAIN, then a request from any client is sent to the servant manager. In this case, the servant manager supports the Servant Locator interface.

A Servant Manager that is behaving as a Servant Activator is responsible for creating an appropriate servant to handle the request. In this way, the servant is created on demand, rather than pre-existing prior to being needed. After the servant is created, it is added to the Active Object

Map to be ready for future requests from the Client. McHale refers to a POA that works this way as a “lazy loader” POA.

A Servant Manager that is behaving as a Servant Locator first calls the CORBA function `preinvoke` on the POA. The `preinvoke` function is responsible for creating the servant. Then the servant manager calls the request on the servant. After the servant call returns, the servant manager calls `postinvoke`.

McHale says that a POA using Servant Manager that is behaving as a Servant Locator can store some, but not all, of the needed servants. McHale says this should be referred to as a “cache” POA since typical use of this would be to have some pre-existing servants to store cached information for a read only database.

Note that the routines `create_reference` and `create_reference_with_id` can create a CORBA object to instantiate an interface without incarnating a servant. The servant could later be incarnated with a servant manager. These two routines require an Interface Repository Identifier as a parameter. The IDL compilers generate these for every item in the IDL file, all type definitions in an IDL file have these identifiers. See Section 8.2.7.2 for a discussion of Interface Repository identifiers.

8.2.8.1.3 Id Uniqueness Policy

Possible values are `UNIQUE_ID` and `MULTIPLE_ID`. The default is `UNIQUE_ID`.

The Id Uniqueness policy says whether or not a servant can incarnate more than one CORBA object that represent interfaces. That is, can a servant serve to implement more than one interface or not?

If `UNIQUE_ID` is set, then a servant can incarnate exactly one interface.

If `MULTIPLE_ID` is set, then a servant can incarnate multiple interfaces (an interface would be associated with a CORBA object).

8.2.8.1.4 Id Assignment Policy

Possible values are

`SYSTEM_ID` and `USER_ID`.

The default is `SYSTEM_ID`.

If `SYSTEM_ID` is set, then the POA assigns an Object Identifier to a CORBA object that represents an interface, when the CORBA object has been activated.

If `USER_ID` is set, then the user gives the CORBA object an Object Identifier.

Normally, a `USER_ID` would only be used when the Lifespan policy is set to `PERSISTENT`. This would typically occur when a persistent object is being used to reference something like a database or a file system.

8.2.8.1.5 Lifespan Policy

Possible values are `TRANSIENT`

and `PERSISTENT`. The default is `TRANSIENT`.

If `TRANSIENT` is selected, then the lifetime of a CORBA object in this POA is the lifetime of the Server. That is, when the Server is terminated, the CORBA object is terminated.

If `PERSISTENT` is selected, then the lifetime of a CORBA object is not associated with the lifetime of the Server. If the Server is terminated and restarted, then the CORBA object is still valid. (This would typically occur when a persistent object is being used to reference something like a database or a file system.)

8.2.8.1.6 Thread Policy

Possible values are `ORB_CTRL_MODEL`,

`SINGLE_THREAD_MODEL`, and `MAIN_THREAD_MODEL`.

The default is `ORB_CTRL_MODEL`.

If `ORB_CTRL_MODEL` is specified, then how the thread model is implemented is up to the ORB vendor, but typically a multithreaded model of some kind is implemented. However, this depends on the ORB. Some possibilities for implementations include: a pool of threads assigned to requests as they come in, a thread per request, etc. The original MICO implementation used a single-thread model, but has been updated with a multithread model. (The single-thread model was also CORBA compliant.)

If `SINGLE_THREAD_MODEL` is used, then requests within a single POA are handled in sequence, but multiple POAs on the Servant obeying the `SINGLE_THREAD_MODEL` are treated concurrently.

If `MAIN_THREAD_MODEL` is used, then requests are handled in sequence across all the POAs on the Servant that obey the `MAIN_THREAD_MODEL` (the main thread of the process handles all requests sequentially across all POAs that obey the `MAIN_THREAD_MODEL`).

8.2.8.1.7 The Implicit Activation Policy: Implicit Activation versus Explicit Activation

Before you can understand the Implicit Activation policy you will have to understand more about what CORBA means by activation, and then we can discuss the pros and cons of *implicit* activation versus *explicit* activation.

Activation means that a servant is associated with (maps to) a CORBA object that represents the interface. The CORBA terminology is that the servant *incarnates* the CORBA object.

The POA is in charge of doing this. Part of the issue here is understanding the different activation mechanisms the POA provides, and why those are important. One method is called *implicit activation*, this is a shorthand that allows the programmer to assume a few defaults. An implicit activation example is shown in Listing 8.37.

In this example, a servant called “myecho” is created first,

then the “_this()” routine is used to perform the implicit activation that maps the “echo_obj” CORBA object representing the interface to the “myecho” servant. The “_this()” routine creates a CORBA object, registers the CORBA object under the POA, and then returns a reference to this CORBA object.

Next the “object_to_string” routine is used to convert the “echo_obj” to a string (stringify the echo_obj reference) that can be written to the “echo.ior” file.

Note that in order for implicit activation to be used, the implicit activation policy on the POA must have previously been set to IMPLICIT_ACTIVATION, that is, to allow implicit activation. The Root POA allows implicit activation by default, but the other POAs default to NO_IMPLICIT_ACTIVATION.

Note that if you’re using implicit activation, the id assignment policy and the id uniqueness policy should normally be set to default values. That is, id assignment is set to SYSTEM_ID, which means that the POA is going to automatically assign an object identifier to the CORBA object when it has been activated. Also, id uniqueness is set to UNIQUE_ID, which means that the servant will map *only* to the current CORBA object (that it was just mapped to, that is, that it just *incarnated*) and not to any other CORBA objects.

The alternative (non-default) setting for the id assignment policy would be USER_ID, which means the user chooses a name (or identifier) for the CORBA objects that are created by the POA. Also, the alternative (non-default) setting for the id uniqueness policy would be MULTIPLE_ID.

One would normally use USER_ID for persistent objects (lifespan policy is set to PERSISTENT) that are meant to live beyond the lifetime of the Client, and one would normally use MULTIPLE_ID for a servant that is meant to map to multiple interfaces (the interface would be associated with a CORBA object). By using implicit activation, we are normally assuming that these are not the case. For implicit activation, we assume that we are using a TRANSIENT object (lifespan

policy is set to TRANSIENT), which means the CORBA object does not live longer than the server object.

LISTING 8.37 Implicit Activation, MICO ORB, in C++

```
// Create a servant
myecho * the_echo = new myecho;

// use implicit activation to map the
CORBA object to the servant echo_var
echo_obj = myecho._this();

// create a stringified object
```

The alternative to implicit activation is *explicit activation*.

There are two routines to do explicit activation: `activate_object(servant name)` and `activate_object_with_id(object identifier, servant name)`.

We've been using explicit activation all along, in all our other CORBA server examples. For example, in the Java Server for the JDK ORB shown in Listing 8.30, we saw:

```
// create servant
myecho the_echo=new myecho();

// get object reference and register it with the ORB
byte [] servant_id =
mypoa.activate_object(the_echo);
```

and in the C++ Server using omniORB from Listing 8.29, we saw:

```
// Create a servant
myecho * the_echo = new myecho;

// use explicit activation to map the CORBA object to the servant
PortableServer::ObjectId_var object_identifier = mypoa->activate_object
(the_echo);
```

In both cases, the *explicit activation* is performed using:

```
activate_object(the_echo);
```

The other method for explicit activation, if it were used instead, would look like, in a Java Server using the JDK ORB:

```
// create servant  
myecho the_echo=new myecho();  
  
// get object reference and register it with the  
ORB byte[] oid = "echo".getBytes();  
byte [] servant_id = mypoa.activate_object_with_id(oid.in(), the_echo);
```

In this case, the *explicit activation* is performed using:

```
activate_object_with_id(oid.in(), the_echo);
```

The explicit activation routines, `activate_object` and `activate_object_with_id`, will both work even if you're using the Root POA, with the implicit activation policy set to `IMPLICIT_ACTIVATION`. This is what we saw in Listings 8.28 and 8.29. So for these routines, the implicit activation policy doesn't really matter. What *does* matter are the id assignment policy, the id uniqueness policy, and the servant retention policy.

Both `activate_object` and `activate_object_with_id` require that the servant retention policy be set to `RETAIN`, or an exception will be thrown. This means that for both these, we must use an Active Object Map in some way. (Note that even if we're using the Active Object Map, depending on what the Request Processing Policy is set to, we could still be using either a default servant or a servant manager.) The `activate_object` routine requires that the Id Assignment policy be set to `SYSTEM_ID` or an exception will be thrown. The `activate_object_with_id` requires that the Id Assignment policy be set to `USER_ID`.

There is an alternative to `_this`, `activate_object`, and `activate_object_with_id`: the routines `create_reference` and `create_reference_with_id` can create a CORBA object to instantiate an interface without incarnating a servant. This servant could later be incarnated with a servant manager. These two routines require an Interface Repository Identifier as a parameter. The IDL compilers generate these for every item in the IDL file: all type definitions in an IDL file have these identifiers. See Section 8.2.7.2 for a discussion of Interface Repository identifiers for interfaces.

In the CORBA examples so far, we've seen only the most basic use of POA policies. In the next two sections, we will see somewhat more advanced examples using some POA policies.

8.2.9 Echo Example—Default Servant

An echo example with the JDK ORB in Java that uses a default servant is shown in Listing 8.37. The first part of this code to note is the following, which sets the POA Request Processing policy to use a default servant:

```
// Create policies for a child POA that uses a default

servant Policy[] thePolicy = new Policy[1];

thePolicy[0] = root_poa.create_request_processing_policy(
    RequestProcessingPolicyValue.USE_DEFAULT_SERVANT
);
```

The next code segment to note is the creation of a child POA that is set with the default servant policy:

```
POA myPOA = root_poa.create_POA("childPOA", null, thePolicy);
```

Next, a name has to be presented to the Naming Service, but this name has to be associated with a CORBA object. Previously, in Section 8.2.6, we had code that created a servant object, and then registered the CORBA object that was associated with that servant on POA, then bound that CORBA object with the name “echo” on the Naming Service.

Now, we don’t want to activate a CORBA object with a servant. We want to create a CORBA object on the child POA, and then bind it to the Naming Service with the name “echo.” However, we don’t want a servant bound to it immediately. What we do instead is the following:

```
org.omg.CORBA.Object echo_ref=
    myPOA.create_reference("IDL:DefaultServantExample:1.0");
```

we previously discussed the `create_reference` routine and the `create_reference_with_id` routine in Section 8.2.8. These two routines require an Interface Repository Identifier as a parameter. The IDL compilers generate these for every item in the IDL file: all type definitions in an IDL file have these identifiers. See Section 8.2.7.2 for a discussion of Interface Repository identifiers for interfaces.

What this routine does, it registers a CORBA object with the child POA. But there is no servant associated with that object. So in the future, when a request comes in from the Client, it will be routed to the default servant.

LISTING 8.38 Echo Example, Default Servant on Server, JDK ORB, in Java

```
import
java.util.Properties;
import
org.omg.CORBA.Object; import
org.omg.CORBA.ORB;
import
org.omg.CosNaming.NameComponent; import
org.omg.CosNaming.NamingContextExt;
import
org.omg.CosNaming.NamingContextHelper; import
org.omg.CORBA.Policy;
import
org.omg.PortableServer.POA; import
org.omg.PortableServer.
```

```

*; import
org.omg.PortableServer.S
ervant;

public class server {

    public static void main(

        String args[] ) { try {

            Properties properties =

                System.getProperties();

                ORB orb = ORB.init(args,

                    null);

                // Get POA and the POA Manager
                org.omg.CORBA.Object
                    the_rootPOA=orb.resolve_initial_references
                    ("RootPOA");
                POA root_poa = POAHelper.narrow(the_rootPOA);

                // Create policies for a child POA

                that uses a default servant Policy[]

                thePolicy = new Policy[1];

                thePolicy[0] =

                    root_poa.create_request_processing_polic
                    y(
                        RequestProcessingPolicyValue.USE_DEFAU
                        LT_SERVANT
                    );
                POA myPOA = root_poa.create_POA("childPOA",
                    null, thePolicy );

                // Create a servant to use as
                the default servant myecho
                servant = new myecho();

```

```
// set the default servant  
myPOA.set_servant(servan  
t);  
  
// Activate the root POA and the child POA  
root_poa.the_POAManager().activate();
```

```

myPOA.the_POAManager().activate( );

// Create a CORBA Object (but don't activate it!!! the default
servant will be used instead)
// This code can use EITHER create_reference_with_id, in
which case you have to give an ID
// name or just create_reference, in which the name is
system generated.

// Note that you MUST use the IDL name format, or this
won't run (it will compile but not run).
// (Note that you MUST have the interface number, but
you could potentially skip the "IDL:", but
    // I don't see any reason why you should do that.)
// The calls create_reference and
create_reference_with_id must pass in an interface
repository ID.

org.omg.CORBA.Object
echo_ref=myPOA.create_reference("IDL:DefaultServant
Example:1.0");

// Find the Naming Service, it's part of orbd
org.omg.CORBA.Object naming_service_ref =
orb.resolve_initial_references ("NameService");

NamingContextExt namingcontext=
NamingContextExtHelper.narrow
(naming_service_ref);

// bind the object reference to the
naming service NameComponent name
= new NameComponent("echo","");
// set name field to "echo" and
kind field to empty string
NameComponent path[] =
{name};

```



```

// Put name in an array of names,
// needed to call the rebind
namingcontext.rebind(path,
echo_ref);

System.out.println("Server is waiting for input");

// Listen for
incoming requests
orb.run();
}

catch (Exception the_exception)
{
    System.err.println("Exception happened, exception is
    "+the_exception);
}
}
}
}

```

LISTING 8.39 Echo Example, Client, JDK ORB, in Java

```

import org.omg.CORBA.ORB;
import
org.omg.CosNaming.NameComn

```

```
import
echo_namespace.echoH
elper; import
echo_namespace.echo;

public class client {
    public static void main(String args[]) {

        try {
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Find the Naming Service, it's part of orbd
            org.omg.CORBA.Object naming_service_ref =
                orb.resolve_initial_references ("NameService");

            NamingContextExt namingcontext=
                NamingContextExtHelper.narrow
                (naming_service_ref);

            // Create a name to be used for searching
            the Naming Service NameComponent
            name = new
            NameComponent("echo", "");
                // set name field to "echo" and
            kind field to empty string NameComponent
            path[] = {name}; // Put name in an array of
            names

            // Find the servant

            org.omg.CORBA.Object proxy =
            namingcontext.resolve(path); echo
            myecho = echoHelper.narrow( proxy );
```

```

// Call the operation on the remote servant
    String return_string=myecho.echostring("hi there");

    System.out.println("");    // print a blank line,
    just for clarity in the output
    System.out.println("The remote servant
    returned: "+return_string);
                                // print the value returned
    from the remote servant System.out.println("");
                                // print a blank line, just for
    clarity in the output
}
catch (Exception the_exception)
{
    System.err.println("Exception happened, exception is
    "+the_exception);
}
}
}

```

For completeness, I've included a Client that can call the Server with the Default Servant. However, it's the same as previous clients that use the Naming Service: compare this to Listing 8.35.

8.2.10 Echo Example—Persistent POA and Default Servant

In this section, we see an example that uses a Persistent POA (and also still uses a default servant).

To set the POA to persistent, we set the Lifespan Policy to PERSISTENT. If PERSISTENT is selected, then the lifetime of a CORBA object is not associated with the lifetime of the Server.

If the Server is terminated and restarted, then the CORBA object is still valid. This would typically occur when a persistent object is being used to reference something like a database or a file system.

When we are using a default servant, it is appropriate to also set ID Uniqueness to `MULTIPLE_ID`. Remember that `MULTIPLE_ID` means that a servant can implement more than one interface (the interface would be associated with a CORBA object). Thus, different interfaces are supported if necessary by this one default servant.

These policies are set using the following code:

```
// Create policies for a child POA that is persistent, maps multiple IDs to the same servant,
// uses a default servant

Policy[] thePolicy = new Policy[3];

thePolicy[0] = root_poa.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);
thePolicy[1] =

    root_poa.create_request_processing_policy(
        RequestProcessingPolicyValue.USE_DEFAULT_SERVANT);

thePolicy[2] =

    root_poa.create_id_uniqueness_policy(IdUniquenessPolicyValue.MULTIPLE_ID); POA
myPOA = root_poa.create_POA("childPOA", null, thePolicy );
```

Otherwise, the example Server in Listing 8.39 is very similar to the example Server in Listing 8.37.

LISTING 8.40 Echo Example, Server with Persistent POA
and Default Servant, JDK ORB, in Java

```
import
java.util.Properties;
import
org.omg.CORBA.Object; import
org.omg.CORBA.ORB;
import
org.omg.CosNaming.NameComponent; import
org.omg.CosNaming.NamingContextExt;
import
org.omg.CosNaming.NamingContextExtHelper; import
org.omg.CORBA.Policy;
import
```

```

// Create policies for a child POA that is
    persistent, maps multiple IDs to the same
    servant,
// uses a default servant

Policy[] thePolicy = new Policy[3];

thePolicy[0] =
root_poa.create_lifespan_policy(LifespanPolicyValue.
PERSISTENT);

thePolicy[1] =
    root_poa.create_request_processing_policy(
    RequestProcessingPolicyValue.USE_DEFAULT_SE
    RVANT);

thePolicy[2] =
    root_poa.create_id_uniqueness_policy(IdUniqu
    enessPolicyValue.
    MULTIPLE_ID);
POA myPOA = root_poa.create_POA("childPOA",
null, thePolicy);
// Create a servant to use as
the default servant myecho
servant = new myecho();

// set the default servant
myPOA.set_servant(servant);
S

// Activate the root POA and
the child POA
root_poa.the_POAManager
().activate();
myPOA.the_POAManager().
activate( );

```

```

// Create a CORBA Object (but don't activate it!!!
the default servant will be used instead)

// This code can use EITHER
create_reference_with_id, in which case
    //you have to give an ID name,
// or just create_reference, in which the name is
system generated.
// Note that you MUST use the IDL name format,
    or this won't run (it will compile but not run).
// (Note that you MUST have the interface
    number, but you could potentially skip the
    "IDL:", but
// I don't see any reason why you should do that.)
// The calls create_reference and
    create_reference_with_id must pass in an
    interface repository ID.
org.omg.CORBA.Object
    echo_ref=myPOA.create_reference("IDL:D
    efaultServantExample: 1.0");

// Find the Naming Service, it's part of orbd
org.omg.CORBA.Object naming_service_ref =
    orb.resolve_initial_references ("NameService");

```

```

NamingContextExt namingcontext=
    NamingContextExtHelper.narrow
    (naming_service_ref);

// bind the object reference to the
naming service NameComponent
name = new
NameComponent("echo","");
// set name field to "echo" and
kind field to empty string
NameComponent path[] = {name}; // Put name in
an array of names, needed to call rebind
namingcontext.rebind(path, echo_ref);

System.out.println("Server is waiting for input");
// Listen for

```

8.3 TECHNOLOGY REVIEW: .NET REMOTING

Microsoft's .NET Remoting allows communication between different processes, whether or not the processes are located on the same computer or different computers across a network. .NET Remoting has been deprecated and is retained for legacy applications. .NET Remoting has been in existence since the .NET Framework release 1.0, back in 2002.

Microsoft currently recommends that new distributed application development should be done using Windows Communication Foundation (WCF) (we will look at WCF in the next section). One major advantage of WCF compared to .NET Remoting is performance. A study by Gupta at MSDN found that WCF is approximately 50% faster than .NET Remoting when comparing the server throughput in terms of number of "operations" per second, across processes on the same machine. Here an "operation" was defined as

request/reply messages with little processing done by the service, while payloads of the messages were 128 bytes, 4 Kbytes, and 256 Kbytes.

Abdul Sami at CodeProject back in 2009 compared the features of .NET Remoting with WCF. He found that in terms of interoperability, .NET Remoting was restricted to a homogeneous environment with both sides in .NET, whereas a WCF hosted server can be accessed by clients written in different languages, and running on many different platforms. He found that .NET supported multiple lower level protocols (HTTP, TCP, and Simple Mail Transfer Protocol [SMTP]), but WCF also supported named pipes and Microsoft Message Queueing (MSMQ) (note that .NET Remoting currently also supports named pipes; we will see an example later on).

8.3.1 .NET REMOTING BASICS

.NET remoting takes place inside the Microsoft Common Language Runtime (CLR), which is the virtual machine used by all .NET routines. CLR implements a program code execution environment which is defined by the Common Language Infrastructure (CLI). The lowest level human readable

language in the CLI is called the Common Intermediate Language, Microsoft's implementation (the Microsoft bytecode) is called the Microsoft Intermediate Language (MSIL). We previously discussed the CLR in Section 6.3.

The .NET framework supports several languages, two of the most common are C# and Visual Basic. The code examples in this textbook will be given in C#.

You might say to yourself, why do I have to learn a whole new language? Doesn't Visual Studio support C++? Note that when you're using C++ in Visual Studio but your code running on top of the CLR, the language you're using is not really C++, it's a different language called C++/CLI that has some similar syntax to C++, but is different in many ways (including garbage collection!). The C++ that was running on the CLR has varied a good bit over the years—originally it was called “managed C++” and was replaced by C++/CLI. Visual Studio does support native C++ but that doesn't run on top of the CLR.

Back in the 2007 timeframe, I wanted to call C# code from native C++. This was so students in our data structures courses, which are taught in C++, could access a robot whose control libraries were written in C# (and which we had need to modify). Over several somewhat painful but interesting days, I was able to make C++ call C++/CLI, and from C++/CLI I could call C#.

Later on, in the 2009–2010 timeframe, one of my PhD students wanted to call the same robot code from Java—he was using a well-known mobile agent framework that was written in Java. We tried all kinds of ways and couldn't get it to work. He got on the Microsoft chat room and asked for help, they told him when we figured it out to please let them know ☺ because they didn't know how. We ended up using sockets to connect from Java to the C# robot code. So the moral of the story is, try to use the languages that the tools expect, whenever possible. It's just easier.

In the .NET paradigm, an application runs inside an “application domain” (“appdomain”). These app- domains allow each application to be isolated from other applications so that a bug in one application can’t affect another application, and similarly, from a security standpoint, malicious code in one app- domain cannot access another appdomain. For this reason, code in one appdomain cannot directly call code in another appdomain.

Note that the same .NET remote call mechanism can be used for a call between two appdomains in the same process, a call between two appdomains on two different processes but on the same computer, or a call between two appdomains on two different computers.

If a client is in the same appdomain as the Server, then it uses a direct reference to the Server. However, if the Client is in a different appdomain from the Server, then it uses a proxy object that represents the remote server object within the Client’s appdomain. The proxy object is generated by .NET using the metadata associated with the remote server (we’ll look at metadata in Section 8.3.1.2). Marshaling of the data is done through this proxy object.

All .NET components and applications run in the CLR. Therefore, windows processes which do not themselves run in the CLR, don’t understand .NET.

However, appdomains run on top of an unmanaged windows process, and are the mechanism through which an unmanaged windows process can access the .NET framework. There may be multiple appdomains on a single unmanaged process.

An appdomain can start up and shut down separately from unmanaged windows process that it runs on. There can be one or more assemblies in a single appdomain. The assemblies inside an appdomain access the .NET services.

If two different appdomains make use of the same class library, that class library is loaded twice.

.NET managed threads are independent of appdomains. If one appdomain creates a thread, that thread can access threads in any other appdomain in the same unmanaged windows process. However, if an appdomain shuts down, any threads inside that appdomain also shut down.

When a .exe file containing a “main” function begins running, a default appdomain is created. The default appdomain has to stick around, you can’t unload it. However, you can create new appdomains on the fly, and you can print out your appdomain’s “friendly name” so you can see which appdomain you’re in. Some code to do this, and the results from running this code are shown in Listing 8.41.

LISTING 8.41 Create AppDomain and Show Which Appdomain You’re In

```
AppDomain
currentAppDomain;
currentAppDomain =
AppDomain.CurrentDomain;
Console.WriteLine(“current appdomain is : ” +
currentAppDomain.FriendlyName); AppDomain
```

8.3.1.2 Assemblies and Metadata

An assembly groups physical files into a single logical unit. A .dll assembly is a “library assembly” and a .exe assembly is an “application assembly.” Usually (and by default in Visual Studio) an assembly has one exe or, alternately, one dll file, but more than one are possible.

When you “add a reference” in Visual Studio, what you are doing is accessing identifier(s) that are defined in a different assembly. In .NET assemblies, there are public components and private components. Only public components may be accessed outside the assembly in which they are defined.

The types in an assembly are defined in the metadata associated with that assembly. This metadata is embedded in the physical file itself (.dll file or .exe file). This metadata is used for “marshaling” in

.NET remoting. That is, .NET remoting reads this metadata to

determine the kinds of data being transmitted and how to format it for transmission and data receipt. When .NET remoting reads this meta- data, it is called “reflection.”

There is a tool called ILDASM provided with the Microsoft SDK that allows you to view the meta- data in an assembly. Figure 8.2 shows the metadata associated with the class shown (later) in Listing

8.45. You can see that the metadata here shows that this class, named HelloWorld.HelloWorld (that is, the class is in a namespace called HelloWorld, and the class itself is named HelloWorld) extends MarshalByRefObject, and that the class contains a method named myHelloWorld which accepts a string parameter and returns a string.

Marshaling in .NET Remoting uses a proxy for an object. The proxy uses the metadata’s exact description of the types and methods in an object to forward the data.

8.3.1.3 Manifest Files

Here we’re going to look at the manifest files that are used in Windows. One reason for telling you what manifest files are is so you don’t confuse them with the metadata. Also, however, the manifest information in an assembly is used to verify version compatibility, and to verify whether or not an assembly is to be trusted. At runtime, .NET verifies that only the referenced assemblies are used, and that only assemblies that employ compatible versions are loaded.

To clarify further, the purpose of the assembly metadata (in terms of .NET remoting) that we saw in Section 8.3.1.1 is to provide a view of the public calls and data in the assembly, so this information

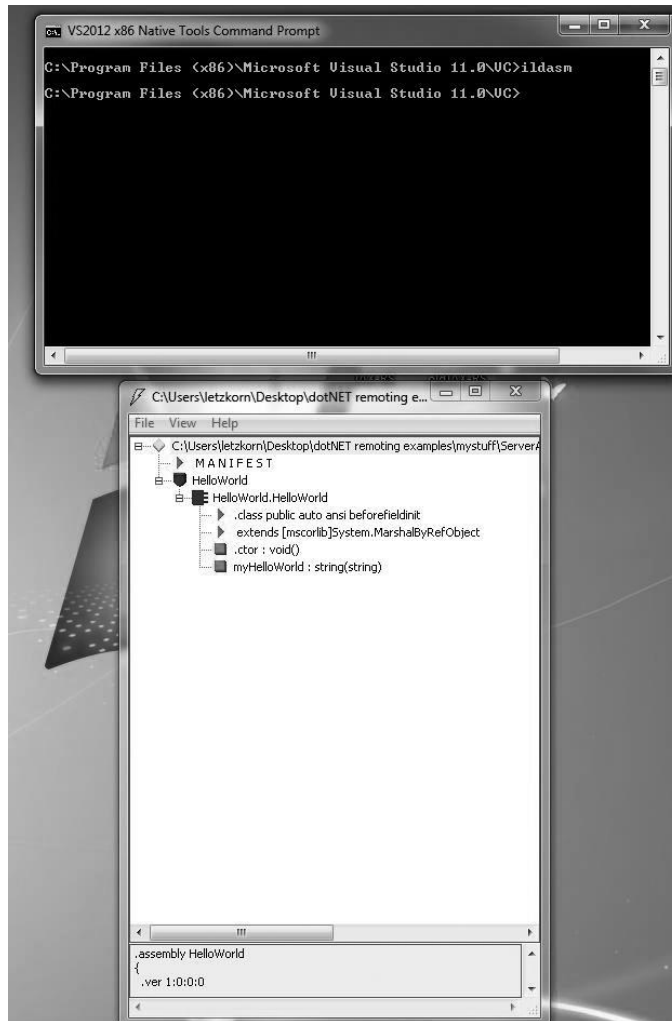


FIGURE 8.2 Using ILDASM to view assembly metadata.

can be used for marshaling (it replaces IDL in the CORBA world, for example). The purpose of manifest files is versioning (prevents .dll hell) and security (makes hacking harder).

There are two different kinds of manifest files in Windows: an “assembly manifest” and an “application manifest.” An assembly manifest tells which files are part of the assembly, how the elements in the assembly relate to each other, all information required to specify version and security and to

resolve references to external entities. An application manifest gives instructions to the operating system when the operating system is loading the application's executable to run it, including which Side-by-Side assembly dependencies the application must bind to at run time, operating system version compatibility, etc. (What is meant by Windows Side by Side was previously discussed in Section 6.2. How applications are stored in a Windows computer was previously discussed in Sections 6.2 and 6.4.)

An example assembly manifest that goes with the HelloWorld.HelloWorld library project from Listing 8.45 (shown later on) is shown here in Figure 8.3. This manifest was read using ILDASM. It tells you the assembly name is HelloWorld and the module name is Hello-World.dll.



FIGURE 8.3 Using ILDASM to view assembly manifest.

You can potentially add to the assembly manifest yourself, for example, “assembly attributes” from `System.Runtime.CompilerServices` or `System.Reflection`.

The `System.Runtime.CompilerServices` namespace is mostly used by compiler writers. However, a class in this namespace called `InternalsVisibleToAttribute` can be used to make an internal type or internal member from an assembly visible to a specific different assembly (a friend assembly).

The `System.Reflection` class can be used to read your assembly’s own metadata to get information about assemblies, modules, parameters, etc., so this information can be used at runtime.

An example application manifest (that goes with the `HelloWorld.HelloWorld` library project from Listing 8.44, shown later on) is shown in Listing 8.42. I used Visual Studio 2012 to generate this manifest file: in Visual Studio 2012, you do Add New Item, and select Application Manifest File. If you

wanted to do this in Visual Studio 2015, then manifest file generation is enabled by default. However, to enable it, select Project Pages, Configuration Properties, Linker, Manifest File, Generate Manifest.

In this file, you can see that to work with different versions of Windows you modify the code appropriately (for example, “If your application is designed to work with Windows 8, uncomment the following...”).

To select who can run the application, the statement:

```
<requestedExecutionLevel level="asInvoker" uiAccess="false" />
```

says that the application will run with the user’s default security levels (actually it really means that it will run using the calling process’ default security levels, but in this case this would be you, because you’re presumably running Visual Studio direct on your account).

To instead force the user to run this application as an administrator, you would change that line to the following line:

```
<requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
```

LISTING 8.42 Example Application Manifest for Code from Listing 8.45

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
xmlns="urn:schemas-microsoft-com:asm.v1"
xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <assemblyIdentity version="1.0.0.0"
name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
```

```
      <requestedPrivileges xmlns="urn:schemas-microsoft-
com:asm.v3">
```

```
        <!-- UAC Manifest Options
```

If you want to change the Windows User Account Control level replace the requestedExecutionLevel node with one of the following.

```
      <requestedExecutionLevel level="asInvoker"
uiAccess="false" />
      <requestedExecutionLevel level="requireAdministrator"
uiAccess="false" />
      <requestedExecutionLevel level="highestAvailable"
uiAccess="false" />
```

Specifying requestedExecutionLevel node will disable file and registry virtualization.

If you want to utilize File and Registry Virtualization for backward

compatibility then delete the requestedExecutionLevel node.

```
-->
```

```
<requestedExecutionLevel level="asInvoker"
      uiAccess="false" />
```

```

    </requestedPrivileges>
  </security>

</trustInfo>

<compatibility xmlns="urn:schemas-microsoft-
com:compatibility.v1">
  <application>
    <!-- A list of all Windows versions that this application
    is designed to work with.
Windows will automatically select the most compatible
environment.-->

    <!-- If your application is designed to work with
Windows Vista, uncomment the following supportedOS
node-->

    <!--<supportedOS Id="{e2011457-1546-43c5-a5fe-
008deee3d3f0}"></supporte- dOS>-->

    <!-- If your application is designed to work with
Windows 7, uncomment the following supportedOS node-
->

    <!--<supportedOS Id="{35138b9a-5d96-4fbd-8e2d-
a2440225f93a}">/>-->

    <!-- If your application is designed to work with
Windows 8, uncomment the following supportedOS node-
->

```

```

<!--<supportedOS Id="{4a2f28e3-53b9-4441-ba9c-
d69d4a4a6e38}"></supportedOS>-->

</application>
</compatibility>

<!-- Enable themes for Windows common controls and
dialogs (Windows XP and later) -->

<!-- <dependency>

<dependentAssembly>

<assemblyI
dentity
type="win
32"
name="Microsoft.Windows.Common-

```

8.3.1.4 .NET Library Files

You should remember our previous discussion from Sections 6.2 and 6.4 about how DLL files are stored, and how the Global Assembly Cache (GAC) works. The .NET Remoting library files (.dll files) are stored in the GAC. I looked at these on my Windows 10 computer, for example, you can see the .dll file for System.Runtime.Remoting in Figure 8.4.

As you can see, these are installed using a strong name for the directory, under the GAC_MSIL directory. See Section 6.4 for a further discussion of how this works.

8.3.2 CALL C# FROM C#, NOT USING REMOTING

Before we get into what .NET Remoting looks like, let's first briefly look at what C# would look like calling a function when

it's *not* using Remoting—just so we can compare ☺ in case you're not all that familiar with C#. In Listing 8.43 we see a class library, and in Listing 8.44 we see the client code to call this class library.

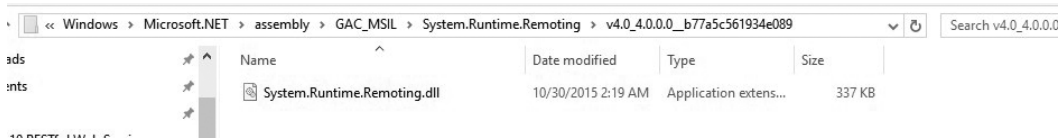


FIGURE 8.4 Windows 10 .NET Remoting files in the GAC.

LISTING 8.43 Basic Class Library, Doesn't Do Much. Should Be Stored in .dll File

```
using System;
using
System.Collections.Ge
neric; using
System.Text;
namespace ClassLibrary
{
    public class CalledClass
    {
        public string CallTheCalledClassObj()
```

LISTING 8.44 Really Basic Client Code, Calls Class Library in Listing 8.43

```
using System;
using
System.Collections.G
eneric; using
System.Text;
using ClassLibrary;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            CalledClass myObj = new
```

Not much to it, right? We create an object of the type of the class in the .dll file, and we call the method on that object.

Pretty basic.

Now let's start adding some .NET Remoting.

8.3.3 .NET REMOTING USING MARSHALING BY VALUE

There are two kinds of Remotable objects in .NET: Marshal by value objects, which are themselves copied and pass from the remote appdomain to the local appdomain, and Marshal by reference objects, in which a proxy object is created in the Client, and the Client accesses the remote object with the proxy object as an intermediary (note that if the Client is in the same appdomain as the marshal by reference object, the Client receives a direct reference to the object rather than marshaling through the proxy).

When you use Marshaling by Value in .NET Remoting, what you mean is that when a client calls a remote object, .NET serializes the object to a stream in the server appdomain, and then deserializes the object in the Client appdomain. Thus, it should be clear that the object must be serializable. The most common reason for marshal by value is to pass a struct (or a class) as a remote parameter.

.NET uses reflection to implement serialization. Note that .NET can also serialize an object to a file. However, when using .NET Remoting the object would be serialized to memory or to a network connection.

An example of a serializable class would be as follows:

```
[Serializable]

public class thisClass
{
    public string myname;

    public int mypublicintegervalue;
    private int myprivateintegervalue;
}
```

Note that when .NET serializes an object, it must serialize the internal structures within the object.

.NET can also serialize more complicated data structures (doubly

linked list, for example).

If a serialized class has some member that is nonserializable then .NET would throw an exception, and you the programmer can write special code to handle the data.

You can use one of the .NET formatters to serialize data. You can do this as part of object serialization for Marshal by Value, or you can do this to write your data in a selected format to a file. You can also do it as part of Marshal by Reference, which we will look at later.

.NET has two default formatters:

1. Binary formatter
2. Simple Object Access Protocol (SOAP) formatter

or you can write your own.

The advantages of a binary formatter are that the data representation is compact and therefore very fast to transmit. The advantages of the SOAP formatter are that it is human readable, platform independent, good for going through a firewall (when using HTTP as the transport protocol). However, it is less compact than the binary representation and therefore slower to transmit. For these reasons, if you're sure you're always going to be using the same platform, and you're not going through a firewall, it would be better to use the binary formatter. By the way, there's an in-depth discussion of SOAP in Section 10.2, as SOAP is also heavily used in non-RESTful web services.

We'll look at these again next when we discuss Marshal by Reference.

8.3.4 MARSHAL BY REFERENCE: THE HEART OF .NET REMOTING

Okay, we've finally arrived at the important .NET Remoting stuff. ☺

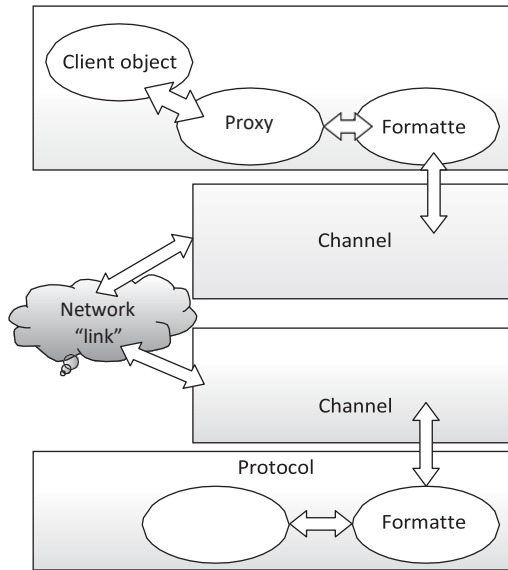


FIGURE 8.5 .NET Remoting architecture diagram.

When we say Marshaling by Reference, we mean that the Client has a proxy object that (together with the formatter and the channel) does the work of marshaling the data to and from the Client. Figure 8.5 shows a diagram of .NET Remoting when Marshal by Reference is being used.

As you can see from this diagram, the channel is responsible for transporting messages between client and server, that is, for actually transmitting the object, while the formatter is responsible for converting the data into the correct format for transmission (serializing the data).

Following the diagram in Figure 8.5, the Client interacts with the remote object through a proxy object. The proxy object is bound to only one remote object, but a client can have multiple proxy objects, each of which accesses the same remote object.

There are two parts to a proxy object: the transparent proxy, which provides the interface of the remote object to the Client (the Client thinks the transparent proxy really is the remote object), and the real proxy which does the data transmission

work.

Still following the diagram in Figure 8.5, when a client makes a call on the proxy, the proxy creates the message object, sends the message object to the formatter, and then blocks, waiting for the call return. The formatter serializes the message object and sends it along to the channel object, which transmits it.

When the message object is received by the channel on the server-side, it forwards the message to a formatter. The formatter deserializes the message and gives it to a server-side entity that is responsible for mapping messages to objects (compare this “server-side entity” to a CORBA POA). The server-side entity constructs a method call that corresponds to the interface of the remote object, then it calls the remote object.

When the remote object finishes handling the method call, it sends the reply back to the same server-side entity that handles objects. The server-side entity creates a returned message object, which is sent to the formatter. The formatter serializes the returned message object and sends it to the server-side channel, which transmits it.

Finally, when the call returns from the channel object, the formatter deserializes the returned message object, and sends the returned message object to the proxy, which was blocked waiting. The proxy returns the returned value and output parameters (that it extracts from the returned message object) to the Client.

Note that if an exception is thrown on the Server, this is serialized and passed back to the client proxy. When the client proxy receives this exception, it re-throws the exception.

You can plug in your protocol to a channel. By default, however, .NET Remoting supports:

- `HttpChannel`
- `TcpChannel`
- `IpcChannel` (named pipes)

Refer to Section 6.5 for a discussion of named pipes. Named pipes were not present in the original .NET Remoting but were added in .NET 2.

An `IpcChannel` (named pipes) is your best choice for interprocess communication on the same computer (that's why it's called IPC—for Inter Process Communication). An HTTP channel is what you need if you're going through firewalls, although it's somewhat slower than the TCP channel since (by default) it uses SOAP, which is XML based, as its data format—the data is not very compact. As you recall from the previous section, the binary format (which the TCP Channel uses by default) is compact and fast to transmit. Note when there is a call between two appdomains in the same process, .NET Remoting employs an internal channel called the `CrossAppDomainChannel` instead of what channels the programmer selected. This is done for performance reasons.

You can change the formatter that you use with a particular channel. For example, the TCP channel uses binary by default, but you can change it to SOAP, and the HTTP channel uses SOAP by default but you can change it to binary. We'll see an example of this later on.

There are three kinds of .NET Marshal by Reference remote objects. (It's a “remote” object because it's “remote” to the Client. It will be running on the Server.):

- Client-Activated
- Server-Activated Single Call

- Server-Activated Singleton

For Client-Activated mode, the remote object on the Server is created when the Client requests it. For the two Server-Activated modes, the Server creates the remote object. When Server-Activated mode is used, the remote objects are also called “well-known” objects (possibly because they can be known by more than one client). Using a client-activated object doesn’t scale well when a large number of clients are accessing the service, since each new client gets a new instance of the remote object, a lot of memory can be used, and each client can potentially hold on to that memory for a long time. If the Client is poorly written, the Client can hold on to the object even when the Client is no longer using it.

For Server-activated mode, you have two options: SingleCall and Singleton. The SingleCall remote object handles only one request at a time. It is instantiated at the beginning of the request and uninstantiated at the end of the request. It does not store any state between method calls (compare this to a stateless session EJB, see Section 8.4.3). This can be a good choice for scalability, since the server-side remote object uses up resources only when it absolutely needs them. The Singleton remote object is instantiated one time, when it is called by the first client that accesses it, and after that the same instance handles requests from several clients. (Compare this to a singleton session EJB, see Section 8.4.4.) A Singleton object doesn’t take up much space, but it may not be good for scalability because if you have a suddenly large number of client requests to handle then that one object could be a bottleneck.

For Client-Activated remote objects, each object is dedicated to the Client that requests it, and different clients get different objects. The lifetime of the object usually is the lifetime of the Client, although a client-activated object could be leased to another appdomain, in which case the lifetime

of the object is extended (until a time specified by a timestamp). Note that the server process must register itself to accept client-activated calls prior to receiving them, and therefore the Server has to be running prior to receiving these requests. It makes sense, by the way, for a Client-Activated remote object to store a state, since the remote object is dedicated to a single client it can hold a running state (compare this to stateful session EJBs, see Section 8.4.5).

For Server-Activated Single Call remote objects, for each new method call a new remote object is created and at the end of the method call the object is uninstantiated. However, the client-side proxy object is not destroyed, so as far as the Client knows, the server-side remote object is there all the time. This means that in some cases the Single Call may have to pretend that it is there all the time—this means it may have to save state. If this is the case, then if, for example, it saves its state in a database,

the connections to the database must
be renewed every time. Every
remote object is associated with a
URL as follows:

- `tcp://localhost:8700`
- `http://localhost:8700`
- When using an `IpcChannel`, you don't specify a port number, just the name of the named pipe:
 - `ipc://servername`

If you're using a Client-Activated object then all you need is the URL. If you're using a Server-Activated object, then you also must specify the name of the remote object:

- `tcp://localhost:8700/remotename`

Both the Client and the Server have to register their associated channels. If using one of

the Server-Activated modes, then Server also has to register the servant object types. The Client has to create a

proxy object of the correct type.

There are two registration methods: programmatic and administrative. Administrative is done through the use of config files. We are going to concentrate on programmatic registration here only because this compares best to the other middlewares we examine in this textbook.

Following are short examples for how to register channels, for each kind of channel: For TcpChannel:

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
IChannel mychannel = new TcpChannel(8700);

ChannelServices.RegisterChannel(mychannel, false);

// false turns off lower level protocol security
```

For an HttpChannel:

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
IChannel mychannel = new HttpChannel(8700);

ChannelServices.RegisterChannel(mychannel, false);

// false turns off lower level protocol security
```

For an IpcChannel (named pipes):

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Ipc;

IChannel mychannel = new IpcChannel("MyChannelName"); // named pipe does not have port #
```

```
ChannelServices.RegisterChannel(mychannel, false);  
  
// false turns off lower level protocol security
```

When a server registers a channel, .NET starts listening for activation requests on a background thread, so your server can continue processing if desired.

It would be a good idea, however, to turn on lower-level protocol security when possible. If this ensured security parameter is true, then if the channel supports security properties then encryption and digital signatures are enabled.

You can only register one channel on a particular port number. Also, you can only register one channel type per appdomain, unless you change the channel name. For example, by default, a TcpChannel is named “tcp,” so to use another TcpChannel you’d have to give it a different name. Similarly for HttpChannel and IpcChannel.

When the Server terminates, the ports and named pipes are automatically freed:

- However, it can be useful to explicitly return resources rather than wait for the garbage collector, so unregister each channel when it’s through:
 - ChannelServices.UnregisterChannel(mychannel);

We will go through several examples in the next sections where we examine the various options of .NET Remoting. We will be using programmatic registration in all cases.

8.3.5 CLIENT-ACTIVATED OBJECT

In this example, we will examine Client-Activated Object mode.

In Listing 8.45 is shown a remote object, which receives a string, prints it’s about to return a Hello- World message, and then returns the HelloWorld message. In the code provided for the instructor with this textbook, this is compiled into a .dll file (class library). The .dll file is then added as a reference to a

separate Visual Studio Server project and to a separate Client project. The Server project and the Client project compile into separate .exe files (Server.exe and Client.exe).

LISTING 8.45 HelloWorld Remote Object in Class Library (.dll File)

```
using System;
using
System.Collections.G
eneric; using
System.Linq;
using System.Text;
using
System.Threading.Tasks;
namespace HelloWorld
{
    public class HelloWorld : MarshalByRefObject
    {
        public string myHelloWorld(string msg)
```

Let's look now at the Server in Listing 8.46. Here, we first register the channel. After that, we register the remote object's name:

```
RemotingConfiguration.ApplicationName = "HelloWorld";
```

And finally we register that we are working in Client-Activated mode, and pass it the remote object we're working with:

```
RemotingConfiguration.RegisterActivatedServiceType( typeof(HelloWorld.HelloWorld) );
```

Then we create a local proxy object:

```
HelloWorld.HelloWorld myobj = new HelloWorld.HelloWorld();
```

LISTING 8.46 Server for Client-Activated Object

```
using System;
using
System.Collections.G
eneric; using
System.Linq;
using System.Text;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
System.Runtime.Remoting.Ch
annels; using
System.Runtime.Remoting.Ch
annels.Http;

using
HelloWorld;
public class
Server
{
```

Now let's examine the Client in Listing 8.47. We first register the channel. Then we set the kind of remote object we plan to activate, and the URL where it is located:

```
RemotingConfiguration.RegisterActivatedClientType( typeof(HelloWorld.HelloWorld),  
    "http://localhost:32000/HelloWorld");
```

And finally we call the proxy object, which will itself, through the magic of .NET Remoting, call the remote object:

```
string returnedvalue = myobj.myHelloWorld(myname);
```

LISTING 8.47 Client for Client-Activated Object

```
using System;
using
System.Collections.G
eneric; using
System.Linq;
using System.Text;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
System.Runtime.Remoting.Ch
annels; using
System.Runtime.Remoting.Ch
annels.Http;

using
HelloWorld;
public class
Client
{
    static void Main(string[] args)
    {
        // Set up the channel, let's use HTTP
        HttpChannel mychannel = new HttpChannel(); //
```

Since this is a Client-Activated object, the object will not actually be created on the Server until the Client calls the proxy object, and the proxy object passes its request message to the Server.

8.3.6 SERVER-ACTIVATED SINGLE CALL

Here we will examine the Server-Activated Single Call mode. In this example, we are again using the remote object from Listing 8.45. Listing 8.48 shows the Server that handles the Server-Activated Single Call mode, while Listing 8.49 shows the Client that goes with this mode.

In Listing 8.48, after the Server registers the channel, note that it registers the channel on port 32000, and it explicitly registers the remote object for access in single call mode:

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof>HelloWorld.HelloWorld),  
    "myHelloWorldObject",  
    WellKnownObjectMode.SingleCall  
);
```

Note that this sets the name of the remote object to "myHelloWorldObject." After this, it waits for calls from the Client.

LISTING 8.48 Server for Server-Activated Single Call

```
using System;
using
System.Collections.G
eneric; using
System.Linq;
using System.Text;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
System.Runtime.Remoting.Ch
annels; using
System.Runtime.Remoting.Ch
annels.Http;
using
HelloWorld;
```

```
        Console.WriteLine("Server is now waiting for calls  
        from client"); Console.ReadLine(); // wait for calls  
    }  
}
```

On the client side, after it registers the channel, the Client explicitly acquires a proxy object, and sets the URL of the proxy at this time, which includes channel 32000 (same port number set by the Server), and uses the same remote object name (“myHelloWorldObject”) that the Server previously used:

```
// Now get a proxy object, this proxy object  
  
// represents the remote object in this local space  
object myproxy = Activator.GetObject(  
    typeof(HelloWorld.HelloWorld),  
    "http://localhost:32000/myHelloWorldObject  
    "  
    );
```

Note that this is very different from what we did in the previous section with a Client-Activated object, where we simply used the “new” operator to create a proxy object (see Listing 8.47).

LISTING 8.49 Client for Server-Activated Single Call

```
using System;
using
System.Collections.G
eneric; using
System.Linq;
using System.Text;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
System.Runtime.Remoting.Ch
annels; using
System.Runtime.Remoting.Ch
annels.Http;

using
HelloWorld;
```

```
// Get the name to send to
the remote
Console.Write("Please enter
your name: "); string
myname=Console.ReadLine(
);

// Call the remote object (call the proxy and the proxy
will call the remote)

// Call the remote object (call the proxy and the proxy
will call the remote)
```

Finally, we call the remote object by calling it on the proxy object:

```
HelloWorld.HelloWorld myHello = (HelloWorld.HelloWorld)myproxy; string
returnedvalue = myHello.myHelloWorld(myname);
```

Note that you have to typecast the proxy for it to match the correct type of the HelloWorld remote object.

Every time this call occurs, the remote object on the Server is instantiated at the beginning of the call, then is executed, and is uninstantiated after it returns the results to the Server channel.

8.3.7 SERVER-ACTIVATED SINGLETON

Here we will examine the Server-Activated Singleton mode. In this example, we are again using the remote object from Listing 8.45. The Server for Server-Activated Singleton is shown in Listing 8.50, and the Client for Server-Activated Singleton is shown in Listing 8.51.

First of all, note that the Client used for Server-Activated Singleton (Listing 8.51) is exactly the same as the Client used for Server-Activated SingleCall (Listing 8.49). I included Listing 8.51 for comparison here (and also for completeness).

LISTING 8.50 Server for Server-Activated Singleton

```
using System;
using
System.Collections.G
eneric; using
System.Linq;
using System.Text;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
```

```

    HttpChannel mychannel = new HttpChannel(32000);
    ChannelServices.RegisterChannel(mychannel, false /*
turn off lower level protocol security */);

    // Register the Singleton object
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(HelloWorld.HelloWorld
        ), "myHelloWorldObject",
        WellKnownObjectMode.Singleton
    );

```

Now let's look at the Server. After we register the channel, we register the Singleton object as follows:

```

// Register the Singleton object
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(HelloWorld.HelloWorld),
    "myHelloWorldObject",
    WellKnownObjectMode.Singleton
);

```

Note that this is almost the same as we did when we registered a SingleCall object in the Server in Listing 8.48. The only difference between that server and this server is that previously we used the parameter;

```
WellKnownObjectMode.SingleCall
```

whereas now we are using the parameter:

```
WellKnownObjectMode.Singleton
```

LISTING 8.51 Client for Server-Activated Singleton

```
using System;  
using  
System.Collections.G  
eneric; using  
System.Linq;  
using System.Text;  
using  
System.Threading Ta
```

```
public class Client
{
    static void Main(string[] args)
    {

        // Set up the channel, let's use HTTP
        HttpChannel mychannel = new HttpChannel();
                                // don't bind the
client channel to a particular port
        ChannelServices.RegisterChannel(mychannel, false );
                                // "false" here is to turn off lower level protocol
                                security

        // Now get a proxy object, this proxy object
        // represents the remote
        object in this local space
        object myproxy =
        Activator.GetObject(
                                typeof(HelloWorld.HelloWorld),
                                "http://localhost:32000/myHelloWorldObject"
                                );

        // Get the name to send to
```

With this mode, only one single remote object is used for all clients.

8.3.8 HOW TO HANDLE DIFFERENT KINDS OF CHANNELS

So far we've been using only HTTP channels. Now we will look at TCP channels and Named Pipes. The code examples provided with the book do these channels as variations on a Server-Activated Singleton example. Here, however, we're

only going to examine the code that is specifically associated with the particular channels.

In Listing 8.52, we see the server-side registration for a TCP channel, and in Listing 8.53, we see the client side registration for a TCP Channel. The only difference between the two is that you can't set the client-side channel on the same port as the server-side channel. I set a separate port number explicitly here, but it would probably be better not to set a particular port number on the client side, because that's more like what we did with the previous HTTP examples.

The code in Listing 8.52 is the only difference required in the server code, compared to an HTTP channel.

LISTING 8.52 Server-Side TCP Channel Registration

```
// Set up the channel, let's use TCP
TcpChannel mychannel = new TcpChannel(32000);
```

LISTING 8.53 Client-Side TCP Channel Registration

```
// Set up the channel, let's use TCP
TcpChannel mychannel = new TcpChannel(8700);
```

Now in Listing 8.54, we see the proxy creation used with a TCP Channel. The only difference between this and the previous Client proxy creation that we saw in Listings 8.49 and 8.51 is the URL parameter. Here we use the URL:

```
"tcp://localhost:32000/myHelloWorldObject"
```

whereas in the previous Client HTTP examples (Listings 8.49 and 8.51), the URL was:

```
"http://localhost:32000/myHelloWorldObject"
```

LISTING 8.54 Client-Side Proxy Creation Using TCP Channel

```
// Now get a proxy object, this proxy object
// represents the remote
// object in this local space
object myproxy;
```

Now let's look at named Pipes. In Listing 8.55, we see the server-side registration for an IPC Channel, and in Listing 8.56, we see

the client-side registration for an IPC channel. Note that the channel doesn't have a port number here, it has a name. The Client channel has a different name from the Server channel.

LISTING 8.55 Server-Side Ipc Channel Registration

```
// Set up the channel, let's use IPC
IpcChannel mychannel = new IpcChannel("thechannel");
    // named pipe uses a channel name instead of a
```

LISTING 8.56 Client-Side Ipc Channel Registration

```
// Set up the channel, let's use IPC
(Named Pipes) IpcChannel
mvchannel = new
```

In Listing 8.57, we see that the Client creates a proxy object similarly to the way it did it in our previous HTTP and TCP examples, only the URL is different. Here, the URL is:

```
ipc://thechannel/myHelloWorldObject");
```

whereas previously the URLs for TCP and HTTP were:

```
"tcp://localhost:32000/myHelloWorldObject"
```

and:

```
"http://localhost:32000/myHelloWorldObject"
```

Note that these differ in “ipc://” versus “tcp://” versus http:// but that also named pipes (IpcChannel) does not use port numbers.

LISTING 8.57 Client-Side Proxy Creation using Ipc Channel

```
// Now get a proxy object, this proxy object
// represents the remote
object in this local space
object mvnproxv =
```

8.3.9 HOW TO HANDLE DIFFERENT KINDS OF FORMATTERS

Now we will look at how to use a different formatter than the default formatter. We will look at an example that uses a binary formatter along with an HTTP channel. As you recall, an HTTP channel defaults to using the SOAP formatter, so we are

replacing the SOAP formatter with the binary formatter. This means that the data will be sent in binary format rather than SOAP format, which as you recall from Sections 8.3.3 and 8.3.4 has advantages in terms of compactness and transmission speed, although of course you give up the SOAP advantage of human readability. You normally wouldn't want to use a binary format if you were going through a firewall, but it can make for very efficient communications inside a company network, or in an embedded system, although more commonly you would choose TCP with its default binary formatter rather than using HTTP and changing its formatter to binary. However, perhaps there are situations where one instance of your code will be going through a firewall while a different instance is totally inside the firewall, such a situation would benefit from this approach because you could use binary format for inside the firewall and SOAP format for through the firewall, but keep the same kind of URL in both cases that employs `http://` as follows:

```
http://localhost:32000/myHelloWorldObject
```

whereas if you used TCP internal you'd have to use "tcp://" for internal but still use http:// for external firewall use, which could perhaps be confusing.

Note, however, that .NET Remoting has been shown to have some issues with firewalls in regard to Network Address Translation (NAT). This is getting a bit advanced for an introductory textbook, so we won't go into that here. However, this is one reason that .NET Remoting has been deprecated in favor of Windows Communication Foundation (or perhaps you could choose a totally different middleware! We're looking at a bunch of them in this textbook.).

LISTING 8.58 Server Side HTTP Channel Registration with Binary Formatter

```
Hashtable settings = new  
Hashtable();  
settings.Add("name",  
"httpbinary");
```

This is done using channel registration. How to do this is shown for the Server side in Listing 8.58, and for the Client side in Listing 8.59.

LISTING 8.59 Client-Side HTTP Channel Registration with Binary Formatter

```
Hashtable settings = new  
Hashtable();  
settings.Add("name",  
"httpbinary");
```

Basically you specify that a BinaryClientFormatterSinkProvider and a BinaryServerSinkProvider will be used (instead of the default

SOAP providers). Note again that the Server and Client have different port numbers.

8.3.10 REMOTE OBJECT LIFETIMES—LEASES ON OBJECTS

We previously talked about remote object lifetimes in Section 8.3.4 when we talked about Client- Activated Objects, Server-Activated Single Call, and Server-Activated Singleton. Remote object lifetimes is really the whole point when we're talking about these modes. With the Client-Activated remote object, the lifetime of the remote object is determined by the Client, while with the Server- activated modes (SingleCall and Singleton), the lifetime of the remote object is determined by the Server.

However, as we will see next, there are some problems with these basic ideas of remote object lifetimes that are handled by the .NET Remoting lifetime lease system. The basic idea behind a lifetime lease is that each remote object is associated with a separate lease object that stores the allowed lifetime of the object. When the lifetime in the lease expires, the remote object is destroyed.

First of all, the lifetime of a Server-Activated SingleCall doesn't participate in the .NET Remoting life- time lease system because the object is recreated for every separate method call and terminated at the end of the method call. Since the lifetime is so well defined, further lifetime management is not necessary.

As you recall, the lifetime of a Client-Activated remote object is the lifetime of the Client: The remote object is created when the Client creates it, and it is destroyed when the Client terminates. But now we ask the question, how does the Server know that the Client has terminated? This is one application of the .NET Remoting lifetime lease system. The lease object associated with the remote object specifies a lifetime, when that lifetime expires the remote object is destroyed.

Also, as we previously discussed, the lifetime of a Server-Activated Singleton remote object is meant to be able to handle all client calls. But suppose there is a very long pause between calls from a client? Should the Server have to have the resources needed for the Singleton remote object tied up for this long period of time? This is another application of the .NET Remoting life- time lease system: because there may be a pause between client calls, the Singleton objects will have an associated lease object that specifies an object lifetime. After that lifetime expires, the Singleton object is destroyed by the Server. If a client calls the Server-Activated Singleton remote object after that, then a new Singleton object is created (together with a new lifetime lease), and the Client won't know the difference between this new Singleton remote object and the previous Singleton remote object.

Now let's examine how the lifetime lease system works. The default initial lease time for a remote object is five minutes (it's stored in a value called "InitialLeaseTime"). It is possible to renew leases. The way this works is as follows:

- *Implicit renewal of lease.* The lease renewal is done automatically when the Client calls the remote object. There is a value called "RenewOnCallTime" that

can be used to specify how long a time period that the lease will be renewed, it defaults to two minutes.

- *Explicit renewal of lease.* This is done with the Renew method on the System.Runtime.Remoting.Lifetime.ILease interface (we will see an example of the code to do this later on).
- *Sponsorship.* The client registers a separate object called a “sponsor” (implements the ISponsor interface) using the ILease interface. Then when the remote object’s lease expires,

the Renewal() method on the sponsor is called, and the sponsor can renew the lease.

Now we will look at an example that illustrates use of the ILease interface, shown in Listing 8.60. I did this by modifying the HelloWorld object, originally shown in Listing 8.44, rather than doing it on the Client, I did this to show that it can be done that way, rather than always having it located in the Client or in the Sponsor. However, I did it as a method on the HelloWorld interface that is called by the Client (this is also shown in Listing 8.60). The activation mode that was used in this example was Server-Activated Singleton.

LISTING 8.60 HelloWorld Remote Object, Uses ILease Interface, and Associated Client Call

```
using System;  
using  
System.Collections.G  
eneric; using  
System.Linq;
```

```
namespace HelloWorld

{
    public class HelloWorld : MarshalByRefObject
    {
        public string myHelloWorld(string msg)
        {
            string mymsg;
            mymsg = "Hello, World, especially to: " +
            msg; Console.WriteLine("about to return
            the Hello World message"); return
            mymsg;
        }

        public override object InitializeLifetimeService()
        {
            // Get a reference to the lease and determine
            if it has been initialized yet ILease
            CurrentLease =
            (ILease)base.InitializeLifetimeService();

            Console.WriteLine("A: At top, current lease state is "
            + CurrentLease.CurrentState); if
            (CurrentLease.CurrentState == LeaseState.Initial)
            {
                CurrentLease.InitialLeaseTime =
                TimeSpan.FromSeconds(3);

                Console.WriteLine("B: After setting initial, current lease
                state is " + CurrentLease.CurrentState);

                CurrentLease.Renew(TimeSpan.FromMillisecond
                s(2000)); Console.WriteLine("C: After setting
```



```

        renew time, current lease state is " +
            CurrentLease.CurrentState);

// Thread.Sleep(10000);
    Console.WriteLine("D: After waiting long enough,
        current lease state is " +
            CurrentLease.CurrentState);
    }

    return null;
}
}
}

```

Client Call to InitializeLifetimeService method:

```

HelloWorld.HelloWorld myHello =
(HelloWorld.HelloWorld)myproxy;
    ... code as before to call the myHelloWorld method...
myHello.InitializeLifetimeService(); // work with the lease

```

The important portion of the HelloWorld code in Listing 8.60 for the purposes of our current discussion is of course the `InitializeLifetimeService` method. It begins by getting a reference to the current lease, using the `ILease` interface (in `System.Runtime.Remoting.Lifetime`):

```
ILease CurrentLease = (ILease)base.InitializeLifetimeService();
```

Next it checks to see if the lease is in its initial state:

```
if (CurrentLease.CurrentState == LeaseState.Initial)
```

The possible states of a lease are as follows:

- Null—error state, not defined
- Initial—beginning state of the lease after the creation of the lease object (to begin with, upon creation of the remote object)
- Active—lease object currently specifies the lifetime of the remote object
- Renewing—the lease has expired, and is in the process of being renewed
- Expired—the lease has expired

Back to the `InitializeLifetimeService` method code: after it is determined that the lease is in initial state, we set the initial lease time to be different from the default—the default is five seconds, we set it to three just to show we can explicitly set the value:

```
CurrentLease.InitialLeaseTime = TimeSpan.FromSeconds(3);
```

Next we'll immediately renew the lease, not because it has expired but just to watch things happen, we're going to explicitly set this to two seconds—this time we're using milliseconds:

```
CurrentLease.Renew(TimeSpan.FromMilliseconds(2000));
```

Notice that we're looking at the state of the lease repeatedly as

we go through this code, for example:

```
Console.WriteLine("B: After setting initial, current lease state is " +  
    CurrentLease.CurrentState);
```

We're going to use these print statements to watch the state of the lease as it changes. Note that the line:

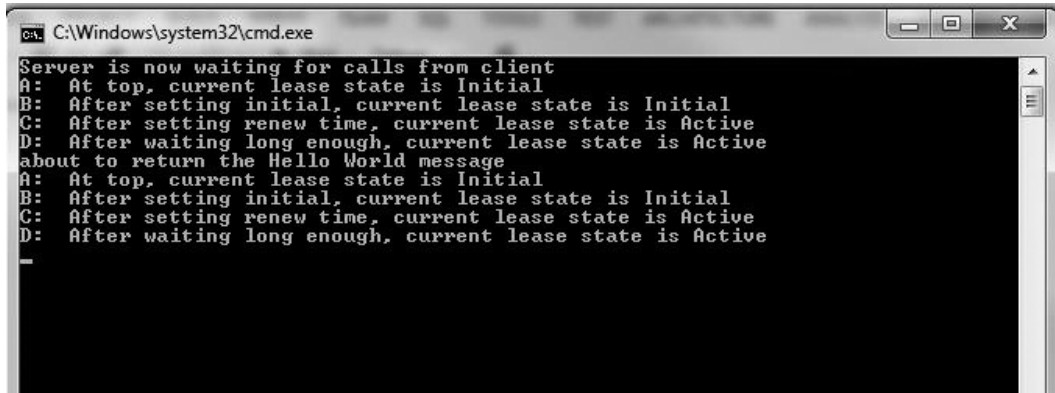
```
Thread.Sleep(10000);
```

has been commented out in this version. What this line is for, is to wait 10 seconds (10000 milli- seconds) to allow the lease to expire so we can see in our print statements that the lease has expired. In the case where we don't have this `Thread.Sleep` line, the lease won't have time to expire before the code stops executing.

Let's look first at the results from this code when the `Thread.Sleep` line is commented out (the lease won't have time to expire). This is shown in the window that was executing the server. See Figure 8.6.

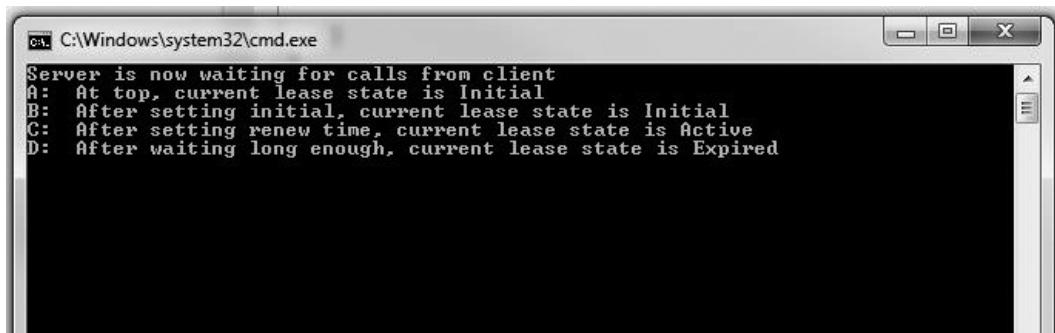
From Figure 8.6, we can see that the lease is originally in Initial State, but moves to Active state after the lease has been renewed.

Now look at Figure 8.7. In Figure 8.7, we see the results from this code when the `Thread.Sleep` line has *not* been commented out, that is, we give the lease time to expire. Again, this is shown in the window that was executing the Server.



```
C:\Windows\system32\cmd.exe
Server is now waiting for calls from client
A: At top, current lease state is Initial
B: After setting initial, current lease state is Initial
C: After setting renew time, current lease state is Active
D: After waiting long enough, current lease state is Active
about to return the Hello World message
A: At top, current lease state is Initial
B: After setting initial, current lease state is Initial
C: After setting renew time, current lease state is Active
D: After waiting long enough, current lease state is Active
```

FIGURE 8.6 Lease without thread. Sleep doesn't have time to expire.



```
C:\Windows\system32\cmd.exe
Server is now waiting for calls from client
A: At top, current lease state is Initial
B: After setting initial, current lease state is Initial
C: After setting renew time, current lease state is Active
D: After waiting long enough, current lease state is Expired
```

FIGURE 8.7 Lease with thread. Sleep had time to expire.

Here we see that we began with the Initial state, after the lease was renewed we moved to Active state. We gave the lease time to expire using the `Thread.Sleep` for 10 seconds, and we can see that the lease has moved to Expired state.

8.3.11 ASYNCHRONOUS .NET REMOTING

In all the examples we've seen so far, we have used synchronous mode. That is, the Client calls the Server and waits (blocks) until the response comes from the Server. However, in many cases it could be helpful for the Client to

continue to do useful work while waiting for the Server's response, especially in cases where the Server might take a long time to come back. This is where asynchronous mode comes in. Asynchronous mode means in general that the Client and the Server execute independently—neither blocks while waiting on the other, they continue to do useful work.

There are two ways this is done in .NET Remoting:

1. *Polling*. The Client calls the Server and continues to do useful work but asks the Server every so often if the Server has completed the Client's response.
2. *Callback function*. The Client calls the Server and tells the Server the name and location of a function located on the Client that the Server can call when the Server completes the Client's response.

8.3.11.1 Asynchronous .NET Remoting—Polling

We will look at the polling example first. I modified the HelloWorld object as shown in Listing 8.61; it now contains a loop with a Thread.Sleep and an occasional print statement so that a fairly long time is passing, and so you can see that the time is passing. The idea here is that it's a good idea for the Client to use asynchronous operation so it can get some work done, because the Server is going to take a long time to finish, and it would be inefficient for the Client to wait for it. Note that the Server itself hasn't changed, it's the same server we saw back in the Singleton example in Listing 8.50.

LISTING 8.61 HelloWorld with a Delay Used to Illustrate Asynchronous Operation

```
using System;
using
System.Collections.Ge
neric; using
System.Linq;
using
System.Text;
using
System.Thread
ing;
using
System.Threading.Tasks;
namespace HelloWorld
{
    public class HelloWorld : MarshalByRefObject
    {
        public string myHelloWorld(string msg)
        {
            string mymsg;

            Console.WriteLine("Inside
```

The asynchronous Client that polls the Server to see if it is done is shown in Listing 8.62. Here a function is defined to wrap the remote method call—this function is called “RemoteAsyncDelegate” and is a “delegate” function. The delegate function must be defined so that it has the same signature, that is, the same parameters and return value, as the method on the remote object that you wish to call. A delegate function has a “BeginInvoke” method and an “EndInvoke” method associated with it. The BeginInvoke method starts the asynchronous call, but it returns immediately and does not block waiting for the call to complete. When the call has eventually completed, the Client will call EndInvoke to retrieve the results of the remote procedure call.

LISTING 8.62 Asynchronous Client—Uses Polling

```
using System;
using
System.Collections.Ge
neric; using
System.Linq;
using
System.Text;
using
System.Thread
ing;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
System.Runtime.Remoting.Ch
annels; using
System.Runtime.Remoting.Ch
annels.Http; using
System.Runtime.Remoting.M
essaging;

using HelloWorld;

public delegate string RemoteAsyncDelegate(string msg);
/* a function to wrap the remote method call, returns a
string because the remote method returns a string */

public class Client
{
    void doWork()
    {
        // Set up the channel, let's use HTTP
        HttpChannel mychannel = new HttpChannel(8700);
        ChannelServices.RegisterChannel(mychannel, false /*
```



```

        turn off lower level protocol security */);

// Now get a proxy object, this proxy object
// represents the remote
object in this local space
object myproxy =
Activator.GetObject(
    typeof(HelloWorld.HelloWorld),
    "http://localhost:32000/myHelloWorldO
    bject"
);

// Get the name to send to
the remote
Console.Write("Please enter
your name: "); string
myname =
Console.ReadLine();

HelloWorld.HelloWorld myHello = new
HelloWorld.HelloWorld(); RemoteAsyncDelegate
thedelegate = new
RemoteAsyncDelegate(myHello.myHello-
    World);

IAsyncResult RemoteResult =
thedelegate.BeginInvoke(myname, null, null);
while(RemoteResult.IsCompleted==false)
{
    Console.WriteLine("Doing useful work while waiting for
    results from server"); Thread.Sleep(1000); // wait one
    second
}

```

```

AsyncResult theResultObject =
    (AsyncResult)RemoteResult; string
    mymsg;

    RemoteAsyncDelegate theDelegate =
        (RemoteAsyncDelegate)theResultObject.Async-
        Delegate;

    mymsg =
        theDelegate.EndInvoke(theResultObject
    )
    Console.WriteLine("HelloWorld: {0}", mymsg);

```

In the Client in Listing 8.62, we first create an instance of the remote object:

```
HelloWorld.HelloWorld myHello = new HelloWorld.HelloWorld();
```

Then we wrap the remote object with our delegate object:

```
RemoteAsyncDelegate thedelegate = new RemoteAsyncDelegate(myHello.myHelloWorld);
```

And then we call the remote object using BeginInvoke:

```
IAAsyncResult RemoteResult = thedelegate.BeginInvoke(myname, null, null);
```

Note that the BeginInvoke returns an IAsyncResult object that we called Remote Object. We use this to check to see if the Server has finished creating the result, as follows:

```
while(RemoteResult.IsCompleted==false)
```

When the result eventually completes, we retrieve our end result using `EndInvoke` as follows;

```
AsyncResult theResultObject = (AsyncResult)RemoteResult;
```

```
string mymsg;
```

```
RemoteAsyncDelegate theDelegate = (RemoteAsyncDelegate)theResultObject.AsyncDelegate;
```

```
mymsg = theDelegate.EndInvoke(theResultObject);
```

```

C:\WINDOWS\system32\cmd.exe

8 File(s)          60,712 bytes
2 Dir(s)  593,935,298,560 bytes free

C:\Users\David\Desktop\Working Code Examples\dotNET remoting examples\mystuff\ServerActivatedSingleton_asynchronous_polling\HelloWorld\Client\Client\Client\bin\Debug>client
Please enter your name: Letha
Doing useful work while waiting for results from server
Inside myHelloWorld, received string :Letha
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
about to return the Hello World message
data received is : Hello, World, especially to: Letha

C:\Users\David\Desktop\Working Code Examples\dotNET remoting examples\mystuff\ServerActivatedSingleton_asynchronous_polling\HelloWorld\Client\Client\Client\bin\Debug>

```

FIGURE 8.8 Client-side results from asynchronous polling example.

The results from running this asynchronous polling example are shown in Figure 8.8. You can see from this that the Server is doing its own work:

Unfortunately, it takes a long time to say hello

At the same time, the Client is continuing independently to do its own useful work:

Doing useful work while waiting for results from server

8.3.11.2 Asynchronous .NET Remoting—Callback Function

In this example, we're going to look at how .NET Remoting can use a callback function for asynchronous operation. As you

recall from Section 8.3.11.2, with a callback function, the Client calls the Server, and tells the Server the name and location of a function located on the Client that the Server can call when the Server completes the Client's response. We will look at two different callback functions here, a simple one with no output parameters, just a return value, and another one that returns output parameters as well as a return value.

In this example, the HelloWorld remote object is the same as in Listing 8.61, it shows that a fairly long time is passing and you can see that time is passing. Again, the Server itself hasn't changed, it's the same server we saw back in the Singleton example in Listing 8.50.

The client code that uses a simple callback function is shown in Listing 8.63. Note that we are still using a Delegate function to wrap the remote procedure.

LISTING 8.63 Asynchronous Client with Simple Callback Function, No Output Parameters

```
using System;
using System.Collections.Generic;

using
System.Linq;
using
System.Text;
using
System.Thread
ing;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
System.Runtime.Remoting.Ch
annels; using
System.Runtime.Remoting.Ch
annels.Http; using
System.Runtime.Remoting.M
essaging;

using HelloWorld;

public delegate string RemoteAsyncDelegate(string msg);
/* a function to wrap the remote method call, returns a
string because the remote method returns a string */

public class Client
{
    [OneWayAttribute]
    public void myCallback (IAsyncResult theResult)
```

```

{
    Console.WriteLine("Top of callback");

    AsyncResult theResultObject =
        (AsyncResult)theResult; string
        mymsg;

    RemoteAsyncDelegate theDelegate =
        (RemoteAsyncDelegate)theResultObject.Async-
        Delegate;

    mymsg = theDelegate.EndInvoke(theResult);

    Console.WriteLine("data received by
        callback is : " + mymsg);

    Console.WriteLine("Bottom of
        callback");
}

void doWork()
{
    // Set up the channel, let's use HTTP
    HttpChannel mychannel = new HttpChannel(); // don't put
    the Client on a particular port
    ChannelServices.RegisterChannel(mychannel, false /* turn
    off lower level protocol
        security */);
    // Now get a proxy object, this proxy object
    // represents the remote object in this local space

```

```

object myproxy = Activator.GetObject(
    typeof(HelloWorld.HelloWorld),
    "http://localhost:32000/myHelloWorldObject"
);

// Get the name to send to the
remote
Console.Write("Please enter
your name: "); string
myname =
Console.ReadLine();

// Call the remote object (call the proxy and the proxy
will call the remote)

// HelloWorld.HelloWorld myHello =
(HelloWorld.HelloWorld)myproxy;
// string returnedvalue =

myHello.myHelloWorld(myname);

HelloWorld.HelloWorld myHello = new
HelloWorld.HelloWorld(); AsyncCallback
theCallback = new
AsyncCallback(this.myCallback);

RemoteAsyncDelegate thedelegate = new

```

In this example, we do the BeginInvoke as follows:

```
HelloWorld.HelloWorld myHello = new HelloWorld.HelloWorld();
```

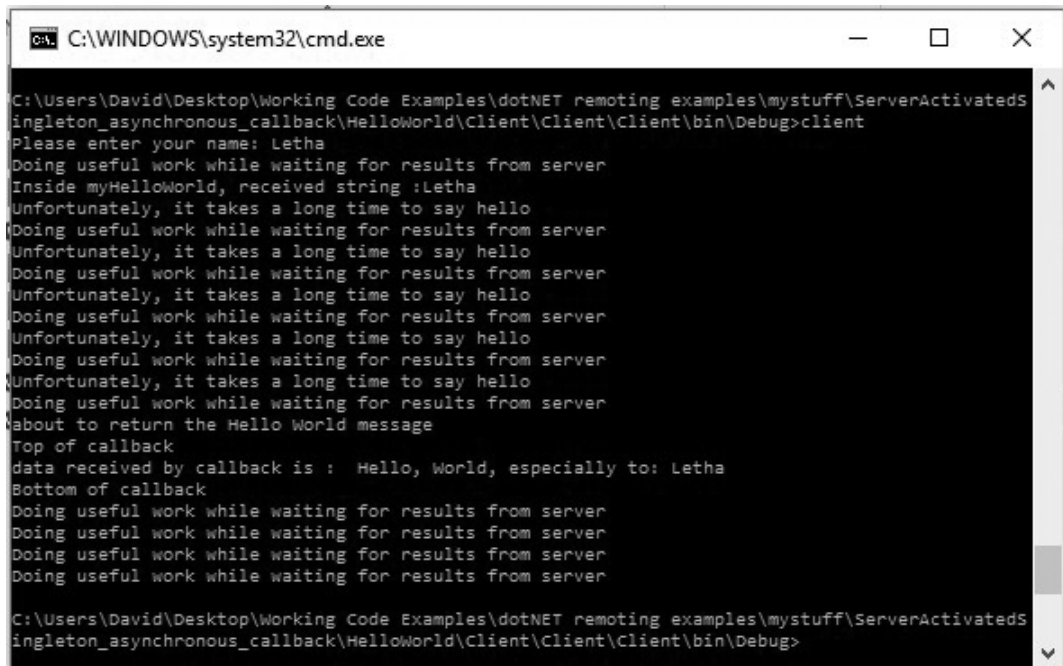
```
AsyncCallback theCallback = new AsyncCallback(this.myCallback);
```



```
RemoteAsyncDelegate thedelegate = new RemoteAsyncDelegate(myHello.myHelloWorld);
```

```
IAsyncResult RemoteResult = thedelegate.BeginInvoke(myname, theCallback, null);
```

As you can see, it's similar to before in the polling example, except that now you create an AsyncCallback function and pass this to BeginInvoke.



```

C:\Users\David\Desktop\Working Code Examples\dotNET remoting examples\mystuff\ServerActivatedS
ingleton_asynchronous_callback\HelloWorld\client\Client\Client\bin\Debug>client
Please enter your name: Letha
Doing useful work while waiting for results from server
Inside myHelloWorld, received string :Letha
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
Unfortunately, it takes a long time to say hello
Doing useful work while waiting for results from server
About to return the Hello World message
Top of callback
data received by callback is : Hello, World, especially to: Letha
Bottom of callback
Doing useful work while waiting for results from server
Doing useful work while waiting for results from server
Doing useful work while waiting for results from server
Doing useful work while waiting for results from server
C:\Users\David\Desktop\Working Code Examples\dotNET remoting examples\mystuff\ServerActivatedS
ingleton_asynchronous_callback\HelloWorld\client\Client\Client\bin\Debug>

```

FIGURE 8.9 Client-side results from simple callback function example.

The call to `EndInvoke` is exactly the same as before in the polling example, except now it is done *inside* the callback function, after the callback is called when the Server has completed its processing:

```

AsyncResult theResultObject = (AsyncResult)theResult;

string mymsg;

RemoteAsyncDelegate theDelegate = (RemoteAsyncDelegate)theResultObject.AsyncDelegate;

mymsg = theDelegate.EndInvoke(theResult);

```

The client-side results are shown in Figure 8.9. As in the previous asynchronous polling example, you can see that the Server and the Client are executing independently, and that the Client continues to execute (doing useful work) while the Server is processing the client request. You can also see when the callback executes, and that the Client continues to work for a while after the callback executes before

it completes.

There is one more item to mention:

```
RemoteResult.AsyncWaitHandle.WaitOne();
```

This statement would cause the Client to block until the remote call finished (the `WaitOne()` function would be called when the remote operation finished, at the same time the callback was called). It didn't actually do anything in this particular case because the callback had finished before the client code got to the `WaitOne()` call. How can I tell? Look at Figure 8.9, it shows the client code continuing to print the statement:

Doing useful work while waiting for results from server

after the following statement had printed:

Bottom of callback

What happened here was that the useful work loop was still executing up until and after the callback ran. It just took long enough for the useful work loop to execute that this was the case. Exercise for the student, ☺ greatly shorten the time in the useful work loop and see when you get to the `WaitOne` call. Try removing the `WaitOne` call and see what happens (if you don't have the `WaitOne` call in that situation, your client will just exit when the loop finishes and you won't have time to do the callback before it quits).

Finally, let's look at the callback example with output parameters. This is shown in Listing 8.64. Your delegate function now has an output parameter, specified with the "out" keyword:

```
public delegate string RemoteAsyncDelegate(string msg, out string extramsg);
```

LISTING 8.64 Asynchronous Client with Callback Function with Output Parameters

```
using System;
using
System.Collections.Ge
neric; using
System.Linq;
using
System.Text;
using
System.Thread
ing;
using
System.Threading.Ta
sks; using
System.Runtime.Re
moting;
using
System.Runtime.Remoting.Ch
annels; using
System.Runtime.Remoting.Ch
annels.Http; using
System.Runtime.Remoting.M
essaging;
```

```
    Console.WriteLine("the extra message was: " +
        extramsg);

    Console.WriteLine("data received by callback is : " +
        mymsg); Console.WriteLine("Bottom of callback");
}
void doWork()
{
    // Set up the channel, let's use HTTP
    HttpChannel mychannel = new HttpChannel(8700);
    ChannelServices.RegisterChannel(mychannel, false
        /* turn off lower level protocol security */);

    // Now get a proxy object, this proxy object
    // represents the remote
    // object in this local space
    object myproxy =
    Activator.GetObject(
        typeof(HelloWorld.HelloWorld),
        "http://localhost:32000/myHelloWorldObject"
    );

    // Get the name to send to
    // the remote
    Console.Write("Please enter
    your name: "); string
    myname =
    Console.ReadLine();

    HelloWorld.HelloWorld myHello = new
    HelloWorld.HelloWorld();

    AsyncCallback theCallback = new
    AsyncCallback(this.myCallback); RemoteAsyncDelegate
```

```

thedelegate = new RemoteAsyncDelegate(myHello.
    myHelloWorld);

string extramsg = "";
IAsyncResult RemoteResult =
    thedelegate.BeginInvoke(myname,out
        extramsg, theCallback, null);

for (int i = 0; i < 10; i++)
{
    Console.WriteLine("Doing useful work while waiting
        for results from server"); Thread.Sleep(1000); //
        wait one second
}
RemoteResult.AsyncWaitHandle.WaitOne();

}
static void Main(string[] args)
{
    Client myClient =
        new Client();
    myClient.doWork()
    ;
}
}

```

When you call `BeginInvoke` you have to give it an extra parameter for the “out” parameter:

```
string extramsg = "";

IAsyncResult RemoteResult = thedelegate.BeginInvoke(myname,out extramsg, theCallback, null);
```

And now when you do the `EndInvoke`, you extract both the out parameter and the return parameter, as follows:

```
string mymsg, extramsg;

RemoteAsyncDelegate theDelegate = (RemoteAsyncDelegate)theResultObject.AsyncDelegate;
mymsg = theDelegate.EndInvoke(out extramsg, theResult);
```

And that’s all there is to it. ☺

8.4 TECHNOLOGY REVIEW: ENTERPRISE JAVA BEANS

The Enterprise Java Bean (EJB) architecture is a Server-side component architecture, and is part of Java EE (Java Platform, Enterprise Edition). EJBs are intended to support several Server-side needs such as integrity of persistence, lifecycle, concurrency, deployment of components, a naming and directory service, asynchronous method invocation, and security, among others. The idea is that since EJBs do this kind of necessary Server-side work, the programmer can concentrate on writing the needed application code (the code for the *enterprise*).

This work is done in an EJB *container* that is implemented by an application server. The EJBs specify how applications (business software) interface to the application server. How the application server does the work is up to the application server.

For the work in this textbook, we are going to use the GlassFish application server.

The EJB specifications since version 3.0, which was released

with Java EE in 2006, are fairly different from the earlier EJB specifications. For example, the Java Persistence API replaces *entity beans* from the earlier versions. We focus on the more recent versions of EJB here.

Note that *Enterprise Java Beans* are quite different from *Java Beans*. A “Java Bean” is a Plain Old Java Object (POJO) that conforms to certain conventions: a public default constructor that has no arguments, get/set accessor functions for data member (attribute) access, required to be serializable. (“Serializable” means to convert an object into a byte stream, so that it can be transmitted or stored, and later on be converted back into a copy of the same object.)

There are two major kinds of EJBs normally discussed:

1. Session Beans
2. Message Beans

Both qualify as EJBs because they both are Server-side components that run in an EJB container in an application server. However, Message Beans operate with a very different paradigm from Session Beans. Message Beans employ a messaging paradigm. Unlike a Session Bean, a Message Bean does not have an interface that a client looks up in order to call a method on the interface. Rather, a Message Bean is not coupled directly to a client. A Session Bean would normally have an interface class and a bean class, whereas a Message Bean only has a Bean class.

In the case of a Message Bean, the Client sends messages to a queue. The Message Bean listens for messages on the queue. Message Beans can support any messaging protocol (queue-based as just described), but by default they support the Java Message Service (JMS). JMS is an example of Message-Oriented Middleware (MOM), and thus Message Beans also are MOM.

In MOM, one or more Producers creates messages and sends them to a queue. Then one or more Consumers retrieve messages from the queue. The queue, however, can actually be

a message broker

that does or can do value-added analyses on the messages, for example, can implement quality-of- service techniques based on priorities. The use of this intermediary message broker (queue) allows the Producers and Consumers to work asynchronously with respect to each other. Also, the use of this intermediary message broker allows loose coupling, which can help when it is necessary to have heterogeneous systems communicate.

Although Session Beans can operate asynchronously as well as synchronously, since the Client calls the interface of the Session Bean, the Session Bean is not an example of MOM.

In this chapter, we are looking at Object-Oriented and Component-Based Middleware, so we will save the Message Beans for Chapter 15, where we examine MOM. (Note, however, that in this text- book we don't focus heavily on MOM, that's just a bit more of an advanced concept, and this is an introductory textbook.)

8.4.1 A BRIEF INTRODUCTION TO SESSION BEAN CONCEPTS

There are three kinds of Session Beans:

1. Stateless Session Beans
2. Stateful Session Beans
3. Singleton Session Beans

A Session Bean can handle a method call from one client at a time.

In this context, "state" means the values of all instance variables inside a Bean at any given time.

In a Stateless Session Bean, the state of the Bean is valid only while the method call from the Client is executing. When the method call from the Client is complete, the state (values of the instance vari- ables) goes away. However, the container can maintain pools of Stateless Session Beans, and the con- tainer can assign a method call coming in from a client to an arbitrary

Stateless Session Bean. In the case of pooled stateless beans, there can be a state across the pool that would apply to any client that is assigned a bean from the pool.

In a Stateful Session Bean, a client and a Stateful Session Bean are considered to be in a “session” (con- versation). When the session (conversation) ends, the state (values of the instance variables) vanishes.

A Singleton Session Bean is shared by many clients. A Singleton Session Bean retains its state for a particular client in between different method calls from that client.

A Stateless Session Bean and a Singleton Session Bean can implement a web service endpoint, but a Stateful Session Bean cannot.

You would choose to use a Stateful Session Bean if you are interacting with a specific client. You would use a Stateless Session Bean to implement a generic task. You would use a Singleton Session Bean if there is a state that must be shared between multiple clients.

There are two ways a Client can acquire a reference to a Session Bean. If the Client is *outside* the Java EE server managed environment, such as Client implemented as a POJO, it must look up the interface by name using the Java directory service, which is accessed using the Java Naming and Directory Interface (JNDI).

If the Client is *inside* the Java EE server-managed environment (such as another EJB, a JAX-RS web service, a servlet, etc.) then the Client may use *dependency injection* (although it can still use JNDI lookup if desired).

A simple definition of *dependency injection* is that it is a way to stuff (inject) a reference to an interface associated with an EJB (right now we’re only considering Session Beans but this can also be used with Message Beans) into an entity within a Java EE server-managed environment (such as another EJB, a JAX-RS web service, a servlet, etc.). This is done using an “@EJB” annota- tion. For example, a servlet could have injected into it (could contain) a reference to the interface

of a Session Bean. A Session Bean could have injected into it (could contain) a reference to the interface of another Session Bean (we'll see an example of this later on).

If the Session Bean has a local interface (according to the annotation) then the local Client that calls that interface must run in the same application, that is, on the same Java Virtual Machine (JVM) as the Session Bean. Local Clients are more efficient than remote Clients, since they do not require marshaling and unmarshaling (serializing) of arguments or transmission of arguments across a network. Local clients employ pass-by-reference, that is, the Session Bean would be working on the same object that the local client is working on, instead of working on a copy of the object (pass-by value), which is what would happen in the case of a remote client accessing a remote interface. (Actual processing of this can depend on the implementation of the application server being used.)

Note that marshaling and unmarshaling in EJBs is less complicated than, say, in CORBA because EJBs are working Java to Java, and Java always stores integers (in the .class files, and inside the JVM) in big-endian order (note that the code actually implementing the JVM operation will store integers in the byte order of the underlying hardware, but this isn't normally seen by the Java programmer). So in case of more complicated data structures such as objects, it's primarily a question of serialization. Object serialization in Java consists of representing an object as a sequence of bytes that includes information about the data type of the object, and about the different data types that make up the object, as well as the data itself. Note that an object is only serializable if the class that defines the object inherits from `java.io.Serializable`.

A Session Bean can have both a remote interface and a local interface. We will see an example of this later on.

A local Client can only be used if the Client and the Session

Beans are packaged in *either* the same jar file or in the same EAR file (or possibly inside a WAR file that is contained in an EAR file). A local Client cannot be used if it is in a web container, and the Session Bean is in a separate container.

A JAR (also sometimes called “jar” after its extension) file (Java Archive) is a type of archive file that is used to collect different Java class files and associated resources of an application (or library) into a single file so it can be deployed.

An EAR file (Enterprise Archive) is a type of archive file that Java EE uses to package multiple modules into a single archive for deployment. It also contains XML-based deployment files that describes how the modules inside the EAR file are to be deployed.

A WAR file (Web Application Archive) is a type of archive file that is used to package a web application. It may contain static web pages, XML files, Java classes, etc.

8.4.2 ENVIRONMENT AND PROCEDURES USED TO BUILD AND RUN EJB EXAMPLES

The examples in this textbook were developed using Java EE 7.

There are several different possibilities for how to build and run EJB examples. You can use an Integrated Development Environment (IDE) such as NetBeans or Eclipse. Each of these has its proponents. You can use the Maven or Apache Ant or Gradle build tools. Each of these has its own set of things you must do in order to get your code to build and run, and each set is different from the other. In an attempt to be as generic as possible, for building these examples I have gone back to very vanilla, plain old Java stuff. I call `javac` directly from the command-line, and I don’t use an IDE or a build tool at all.

And of course there are different application servers. I chose GlassFish because it’s the Java EE reference implementation. To develop my examples, I downloaded the Java EE 7 and GlassFish bundle.

To compile an interface:

```
export GLASSFISH_HOME= ..put your stuff here...
```

```
export JAR_FILE_ONE=$GLASSFISH_HOME/glassfish/modules/javax.ejb-api.jar javac -  
classpath ".:$JAR_FILE_ONE" -d . HelloWorld.java
```

To compile a bean:

```
export GLASSFISH_HOME=..put your stuff here...  
  
export JAR_FILE_ONE=$GLASSFISH_HOME/glassfish/modules/javax.ejb-api.jar javac -  
classpath ".:$JAR_FILE_ONE" -d . HelloBean.java
```

Then create a jar file containing HelloWorld.class (the interface) and HelloBean.class (the Stateless Session bean):

```
jar cvf helloworld.jar example
```

Here, “example” is the directory that was created when javac was run. This is because the package that HelloBean is in was defined as follows:

```
package example.helloworld;
```

To compile a POJO client:

```
javac -classpath . -d . HelloClient.java
```

HelloBean also has to run *inside* a container in an application server. We’re using the GlassFish application server for these examples. To do this, start GlassFish. Then either use the GlassFish admin console or else the GlassFish command-line to deploy this jar file. How to start GlassFish and deploy applications such as EJBs or servlets on GlassFish is described in Section 4.11.2.1.

Note also that the code provided for the instructor with this textbook includes detailed instructions for how to compile all the EJB examples.

8.4.3 STATELESS SESSION BEANS

In this section, we examine Stateless Session Beans. Recall from our earlier discussion, that with a Stateless Session Bean,

the state of the Bean is valid only while the method call from the Client is executing. When the method call from the Client is complete, the state (values of the instance variables) goes away. However, the container can maintain pools of Stateless Session Beans, and the container can assign a method call coming in from a client to an arbitrary Stateless Session Bean. In the case of pooled stateless beans, there can be a state across the pool that would apply to any client that is assigned a bean from the pool.

8.4.3.1 Very Simple Stateless Session Bean Example

First we will look at a very simple Stateless Session Bean example.

In Listing 8.65 we see a remote interface—it is called a *remote* interface because it is annotated with “@Remote.” This means that any data that is sent to or received from this interface would be marshaled and unmarshaled (serializing) for transmission across a network (we are assuming distributed objects here).

In Listing 8.66 we see a very simple Stateless Session Bean. This defines a class called “HelloBean” that implements the HelloWorld Interface from Listing 8.65. This is pretty well just like any Java class implementing a Java interface. The difference in Listing 8.66 is that the HelloBean class is annotated with “@Stateless” (it has to import `javax.ejb.Stateless` to use this annotation) and that HelloBean implements an interface that was described as “@Remote.”

Note that partly because of these annotations, HelloBean becomes a Stateless EJB and not just a POJO. Also, in order to be an EJB, HelloBean and the interface file also each have to be

linked with the `javax.ejb-api.jar` file. And then the jar file containing the `HelloBean` has to be deployed and run in a container, such as the `GlassFish` container.

LISTING 8.65 HelloWorld Interface

```
package
example.helloworld; import
javax.ejb.Remote;
@Remote
```

LISTING 8.66 Simple Stateless Bean

```
package
example.helloworld;
import
javax.ejb.Stateless;
@Stateless
public class HelloBean
```

In Listing 8.67, we see a Client call the simple Stateless Bean, this Client is implemented as a POJO. Since this Client is a POJO, the Client must use the JNDI to find the location of the Stateless EJB that it is calling.

The JNDI maps names to object locations. Typically, a name is simply a string.

JNDI names are organized into name hierarchies (similar to file directory systems). The base of this name hierarchy is called an *initial context*. When a Client passes a name to JNDI for lookup, JNDI starts at the initial context and looks down the name hierarchy to search for the matching name.

To find the initial context, we do:

```
Context ic= new InitialContext();
```

To look up a name (in this case, our HelloBean in the helloworld package) in this particular context, we do:

```
Object object_ref = ic.lookup("java:global/helloworld/HelloBean");
```

Note that we are using the *java:global* namespace here. Names in the java:global JNDI namespace are accessible to any application.

Finally, we must create an appropriate proxy object to represent the HelloWorld interface in the Client. We do this via a downcast from the object reference returned from JNDI:

```
HelloWorld proxy= (HelloWorld) PortableRemoteObject.narrow(object_ref,HelloWorld.class);
```

Finally, we call the remote procedure on the remote EJB using this proxy:

```
String returned_string = proxy.thehello();
```

Since the HelloWorld interface (see Listing 8.65) was annotated as “@Remote”, when this call on the proxy is performed, parameters to the EJB (in this particular case there weren’t any parameters in the “thehello()” call) and the returned function value from the EJB will be marshaled and unmarshaled (serialized) and also, in the case where the HelloBean is located on a different computer than the Client, transmitted across a network.

LISTING 8.67 POJO Client That Calls the Simple Stateless Bean

```
package
example.helloworld;
import
javax.naming.InitialC
ontext; import
javax.naming.Context
;
import
javax.naming.NamingEx
ception; import
javax.rmi.PortableRemote
Object;

public class HelloClient {
    public static void main
        (String[] args) { try {

        Context ic= new InitialContext();

        // Names in the java:global JNDI namespace are
        // accessible to any application.
        // (This is how EJB references may be accessed
        // outside the application where
        // they're defined.)
        Object object_ref =
        ic.lookup("java:global/helloworld/HelloBean");
```

8.4.3.2 Very Simple Stateless Session Bean Example Passes an Array Parameter

In this section, we will look at how an array can be passed as a parameter to a Remote Stateless EJB. Remember that in EJB, all parameters passed to a remote interface are “passed by value” in that they are marshaled (serialized) for transmission and unmarshaled (deserialized) at the receiving end.

In this case, we are passing a byte array, see Listing 8.68. Note that unlike, say, CORBA, EJBs employ the word “byte” and not “octet.” This is because with EJBs, you most commonly expect Java to be running on both ends, and Java uses the word “byte,” so there is usually no confusion. (Note that EJBs since EJB 2.1 in 2003 can have their interfaces exposed as web services, so invocation of these interfaces using other programming languages is possible, but the traditional expectation using EJBs is that Java would be running on both sides.)

LISTING 8.68 HelloWorld Interface with Array Parameter

```
package  
example.helloworld;  
import  
javax.ejb.Remote;  
import java.io.*;
```

The Stateless HelloBean in Listing 8.69 that handles the interface from Listing 8.68 is annotated with “@Stateless” but after that there’s nothing special about the Java code that handles the interface; it could be used for any interface that happens to pass an array, either local or remote. In this case, it extracts the bytes from the byte array and appends them to a “Hello, values received are ” string, and then returns that string to the Client.

LISTING 8.69 Simple Stateless Bean with Array Parameter

```
package
example.helloworld;
import
javax.ejb.Stateless;

@Stateless
public class HelloBean
    implements HelloWorld {
    public String thehello(byte[]
myarray) {

        int arrayLength=myarray.length;
        String mvreturnString="Hello.
```

In Listing 8.70, we see only the code required to set up a byte array and send it as a parameter to the proxy of the interface from Listing 8.68. Otherwise, this code is the same as the POJO Client earlier shown in Listing 8.67.

**LISTING 8.70 Code Snippet from POJO Client That Calls
Simple Stateless Bean with Array Parameter**

```
// call the remote method  
on the local proxy byte[]
```

8.4.3.3 Very Simple Stateless Session Bean Example Passes an Object Parameter

In this section, we will look at how an object (of a particular class type) can be passed as a parameter to a Remote Stateless EJB. Remember that in EJB, all parameters passed to a remote interface must be marshaled and then unmarshaled.

The interface that sends an object is shown in Listing 8.71. Note that we have defined the class definition in this file as well. For an object of a class to be sent to a Remote interface, the class must be *serializable*. We previously discussed what that means back in Section 8.4.1. In terms of implementation, it means that the class must implement *java.io.Serializable*.

In this case, just for the object to have something to do, class A defines a private variable called *j*, and get and set accessor functions to manipulate *j*.

LISTING 8.71 HelloWorld Interface with Object Parameter

```
package
example.helloworld;
import
javax.ejb.Remote;
import java.io.*;

class A implements
    java.io.Serializable {
    int j;

    public void
        setj(int val)
        { j=val;
        }
}
```

The Stateless HelloBean in Listing 8.72 that handles the interface from Listing 8.71 is annotated with “@Stateless” but after that there’s nothing special about the Java code that handles the interface, it

could be used for any interface that happens to pass an object that instantiates Class A, either local or remote. In this case, it uses an accessor function to get the value of *i*, and returns a message related to whether *i* is positive or negative.

LISTING 8.72 Simple Stateless Bean with Object Parameter

```
package
example.helloworld;
import
javax.ejb.Stateless;
@Stateless
public class HelloBean
    implements HelloWorld { public
    String thehello(A myobj) {
        int
```

In Listing 8.73, we see only the code required to set up an object of type Class A and send it as a parameter to the proxy of the interface from Listing 8.71. In this case, it uses the set accessor function to set the value of *i* to a negative number. Otherwise, this code is the same as the POJO Client earlier shown in Listing 8.67.

LISTING 8.73 Code Snippet from POJO Client That Calls Simple Stateless Bean with Object Parameter

```
// call the remote method
on the local proxy A myobj
```

In this example, in addition to basic HelloWorld interface that is the same as the interface seen earlier in Listing 8.65, there is one interface that allows remote access (annotated by “@Remote”), shown in Listing 8.74, and another interface that allows local access (annotated by “@Local”), shown in Listing 8.75.

Note that the interface in Listing 8.74 is the same as the interface in Listing 8.65. Having a separate remote interface like this allows us to easily illustrate dependency injection from within the HelloBean that implements the interface in Listing 8.65. We’ll talk more about this shortly.

LISTING 8.74 HelloWorld Remote Interface

```
package
example.helloworld;
import
javax.ejb.Remote;
```

LISTING 8.75 HelloWorld Local Interface

```
package
example.helloworl
d; import
javax.ejb.Local;
```

The Client used here to call the base interface (from Listing 8.65) is a POJO client that is the same as the Client seen earlier in Listing 8.67. This Client uses JNDI to call the basic HelloWorld interface from Listing 8.65, which it must do because it is a POJO. We are going to look at *dependency injection* here, but as we discussed earlier in Section 8.4.1, injection can only be used if the Client is *inside* the Java EE server-managed environment (such as another EJB, a JAX-RS web service, a servlet, etc.), then the Client may use “dependency injection” (although it can still use JNDI lookup if desired). This is done using an “@EJB” annotation.

So in order to demonstrate dependency injection, we define a HelloBean that implements the basic HelloWorld interface class. Then from inside this HelloBean we employ dependency injection (instead of the alternative, using JNDI) to provide references to the remote interface, and separately to the local interface. Then from inside this HelloBean we perform a remote procedure call on the remote interface, and a local

procedure call on the local interface. That is, although this HelloBean implements the basic interface as a session bean and is called by an external POJO client, this HelloBean also is serving as a EJB client to call other EJB session beans. This HelloBean is shown in Listing 8.76.

We will also see a HelloBean (HelloBeanRemote) that implements the remote interface, and a HelloBean (HelloBeanLocal) that implements the local interface. HelloBeanRemote is shown in Listing 8.77, and HelloBean Local is shown in Listing 8.78.

Let us first examine the dependency injection in Listing 8.76. There are two examples here, in both cases, this is performed using the “@EJB” annotation. First, an injection of the remote interface:

```
// Inject the Remote interface
@EJB

private HelloWorldRemote myremotebean;
```

And second, an injection of the local interface:

```
// Inject the Local interface
@EJB

private HelloWorldLocal mylocalbean;
```

What this does in each case is create a proxy object that contains the appropriate location (address) information. The procedure calls to these proxy objects are very similar. Here is a remote procedure call to the proxy object that represents the remote interface:

```
// Call the Remote interface
String returned_valueA;

returned_valueA = myremotebean.thehello();
```

And here is a local procedure call to the proxy object that represents the local interface:

```
// Call the Local interface
String returned_valueB;

returned_valueB = mylocalbean.thehello();
```

In Section 8.4.1, we previously discussed the difference between a local interface and a remote interface. Remember that if the Session Bean has a local interface (according to the annotation) then the local Client that calls that interface must run in the same application, that is, on the same JVM as the Session Bean. Local Clients are more efficient than remote Clients, since they do not require marshaling and unmarshaling (serializing) of arguments or transmission of arguments across a network. (Actual processing of this can depend on the implementation of the application server being used.)

However, although a Client calling local interface must be in the same application as the Session Bean called, it is possible for a Client to use a remote interface in the same application as

the Session Bean that is called—that is what we have seen here.

LISTING 8.76 HelloBean Performs Dependency Injection

```
package
example.helloworld;
import
javax.ejb.EJB;
import javax.ejb.Stateless;
import
example.helloworld.HelloBean
Remote; @Stateless
    public class HelloBean implements HelloWorld {

    // Inject the
    Remote interface
    @EJB
    private HelloWorldRemote myremotebean;
```

```

        returned_valueA = myremotebean.thehello();

        // Call the Local
        interface String
        returned_value
        B;
        returned_valueB = mylocalbean.thehello();

        String value to return:

```

Note that the `HelloBean` that implements the `HelloWorldRemote` interface is almost the same as the one previously shown in Listing 8.66. Again, the purpose of using this here is to allow the `HelloBean` in Listing 8.76 to illustrate dependency injection of a Remote Session Bean.

LISTING 8.77 `HelloBeanRemote` Implements Hello Bean Remote Interface

```

package
example.helloworld;
import
javax.ejb.Stateless;
@Stateless

```

In Listing 8.78, the notation “`@LocalBean`” is used. We really didn’t need a separate interface such as the one we used in Listing 8.75 because this annotation can both define the interface and the bean that implements the interface (if we weren’t using the separate interface we would leave off the “implements” clause). We could instead have annotated the class as “`@Local`” or alternately annotated the class as `@Local(LocalHelloBean.class)`. Or if we hadn’t been using a separate interface, and hadn’t annotated this class at all, it would have defaulted to local. It’s generally best to be explicit

about whether an interface (or class) is local or remote. And defining separate interfaces is a good way to describe and clarify the code, so that should be what you normally do.

LISTING 8.78 HelloBeanLocal Implements Hello Bean Local Interface

```
package  
example.helloworld;  
import  
javax.ejb.LocalBean;
```

```
public String thehello() {
    return "Hello from HelloBeanLocal";
}
```

8.4.3.5 Simple Stateless Session Bean Example That Illustrates Stateless Characteristics

This is an example that illustrates that a Stateless Session Bean really is Stateless. To do this, we see a slightly more complicated interface than what we've been using so far; see the interface in Listing

8.79. This one implements getcount and setcount accessor functions, and an increment count function, in addition to the original thehello() function.

The HelloBean shown in Listing 8.80 implements this interface. It's more complicated than the original Stateless HelloBean from Listing 8.66 because it has to implement the additional functions. However, it is still very simple, as you can see.

LISTING 8.79 Slightly More Complicated HelloWorld Interface—getcount, setcount, incrementcount

```
package
example.helloworld;
import
javax.ejb.Remote;
@Remote
```

LISTING 8.80 Stateless HelloBean Implements Slightly
More Complicated HelloWorld Interface

```
package
example.helloworld;
import
javax.ejb.Stateless;
@Stateless
public class StatelessHelloBean
    implements StatelessHelloWorld {
    private int count=0;
    public String thehello() {
        return "Hello to all the World!!!!".
```

```
        return "set the stateless counter";
    }
    public String
        incrcount() {
        count++;
        return "incremented the stateless counter";
    }
}
```

The Client shown in Listing 8.81 is where most of the work is done to illustrate that the HelloBean from Listing 8.80 really is stateless. Remember that the application server retains a pool of Stateless Beans. What this Client will show is that the initial value returned will be the value from the most recent Stateless Bean used by *any* Client proxy object, rather than the value that was previously used by the *same* Client proxy object that is currently being used. That is to say, the Stateless HelloBean is not retaining a conversational state with each Client proxy object that is used to call it.

To clarify a little more: the Client in Listing 8.81 creates two proxy objects: Proxy A and Proxy B. Proxy A does a remote procedure call on the HelloBean, reads the initial value of count, sets its count to 3 and increments it to 4 (this is done in a subroutine on the Client named *calltheremote*). Then Proxy B does a remote procedure call on the HelloBean, reads the initial value of count, sets its count to 100 and increments it to 101. Finally, Proxy A does another remote procedure call on the HelloBean, reads the initial value of count, sets its count to 7 and increments it to 8. (Note that I could have left out the increment count and done this just with *getcounts* and *setcounts*, but I thought a little bit more EJB examples wouldn't hurt. ☺)

Finally, at the end Proxy A does a *getcount* (get the value) and Proxy B does a *getcount* (get the value) on the HelloBean, one after the other.

The output from a run of this Client is shown in Table 8.1.

Note that the initial value of Proxy B is 4, which is the final value from Proxy A. Also, note that the initial value for the second set of calls by Proxy A is 101. If a conversation were being maintained

TABLE 8.1
Output from a Run
Illustrating Statelessness of
the Stateless Bean

	Count Value
Proxy A initial value	8*
Proxy A set value	3
Proxy A value incremented to	4
Proxy B initial value	4
Proxy B set value	100
Proxy B value incremented to	101
Proxy A initial value	101
Proxy A set value	7
Proxy A value incremented to	8
Final Proxy A value	8
Final Proxy B value	8

using the call from the Client on Proxy A, then when Proxy A was used a second time, the initial value would have been 4 and not 101.

If Proxy A and Proxy B had separate conversations going, and the state was maintained by the HelloBean for each proxy between separate sets of calls from each Proxy, then Proxy A would get the value that Proxy A set on the HelloBean, and the fact that Proxy B set another value on the HelloBean *after* Proxy A's first set of calls and before Proxy A's second set of calls wouldn't matter.

What is clearly happening here is that the container running on the application server is returning from the pool the most recent bean used, regardless of which Proxy object had just been used to access it, and it retains the value from the most recently completed call, until it is explicitly set on the current set of calls.

Note that the very first initial value for a Proxy A call contained an 8. This is marked with an asterisk, because what really happened here is that this was the second time I had run this code, so the most recent object available in the pool contained the 8 that was the value from the first run I did.

Conclusion: The Stateless Hello Bean is really stateless.

LISTING 8.81 Client That Illustrates Statelessness of Stateless Session Bean

```
package
example.helloworld;
import
javax.naming.InitialC
ontext; import
javax.naming.Context
;
import
javax.naming.NamingEx
ception; import
javax.rmi.PortableRemote
Object;

public class StatelessHelloClient {
    public static void main
        (String[] args) { try {
            Context ic= new InitialContext();

            // Get one proxy object
            Object object_ref =
                ic.lookup("java:global/statelesshelloworld/Sta
tlessHelloBean");
```

```
System.out.println("Now get the count  
from original proxy"); int  
count_returned = proxyA.getcount();  
System.out.println("proxyA count is  
"+count_returned);
```

```
System.out.println("Now get the count  
from the second proxy");  
count_returned = proxyB.getcount();  
System.out.println("proxyB count is  
"+count_returned);
```

```
call_the_remote(proxyA, 7); // call all the methods  
on the proxy
```

```
System.out.println("Now get the count  
from original proxy"); count_returned  
= proxyA.getcount();  
System.out.println("proxyA count is  
"+count_returned);
```

```
System.out.println("Now get the count  
from the second proxy");  
count_returned = proxyB.getcount();  
System.out.println("proxyB count is  
"+count_returned);
```

```
System.out.println("When I ran this,  
whatever the last value was");  
System.out.println("proxyA got it and  
proxyB also got it");  
System.out.println("The EJB is stateless  
so it's not really");  
System.out.println("retaining its state
```



```

        between method calls.");
        System.out.println("What happens is the
        container is repeatedly");
        System.out.println("giving back the same
        bean from the pool.");
        System.out.println("So it gets rewritten
        every time.");
    }
    catch (NamingException ex) {
        // You might get a naming exception if you forgot to
        // deploy the jar file
        // You might get a naming exception if you forgot to
        // start GlassFish System.out.println("Naming
        exception");
        System.out.println("cause
        is"+ex.getCause());
        System.out.println("stack
        trace is ");
        ex.printStackTrace();
    } catch (Exception ex) {
        System.out.println("gene
        ral exception");
    }
} // end Main()
public static void call_the_remote(StatelessHelloWorld
proxy,int value) {
    // call the remote method
    on the local proxy String
    returned_string =
    proxy.thehello();
    System.out.println(returne
    d_string);

    int count_returned =
    proxy.getcount();
    System.out.println("first value of
    count is "+count_returned);
}

```

```
returned_string =  
proxy.setcount(value);  
System.out.println(returned  
_string);  
  
count_returned = proxy.getcount();  
System.out.println("count is now  
"+count_returned);
```

```
returned_string =  
proxy.incrcount();  
System.out.println(returned_string);  
  
count_returned = proxy.getcount();  
System.out.println("count is now
```

8.4.4 SINGLETON SESSION BEANS

In this section, we examine Singleton Session Beans. You should recall from Section 8.4.1 that a Singleton Session Bean is shared by many clients, and a Singleton Session Bean retains its state for a particular client in between different method calls from that client.

In Listing 8.82, we see a HelloWorld interface that also includes `getvalue` and `setvalue` accessor functions (they access an integer).

LISTING 8.82 HelloWorld Interface Including `getvalue` and `setvalue` Accessor Functions

```
package  
example.helloworld;  
import  
javax.ejb.Remote;  
@Remote  
public interface
```

In Listing 8.83, we see a Singleton Bean, annotated with “`@Singleton`,” that implements the interface in Listing 8.83. Note that this example is also annotated with “`@Startup`.” What this means is that the container in GlassFish must immediately create the `HelloBean` when the application runs. If you don’t

specify “@Startup” then its up to the container when to create the HelloBean object.

LISTING 8.83 Simple Singleton Session Bean

```
package  
example.helloworld;  
import  
javax.ejb.Startup;  
import  
javax.ejb.Singleton  
• @Startup
```

```
}  
public void  
    set_value(int  
    i) { value=i;  
}  
  
public int  
    get_value(  
    ) { return
```

In Listing 8.84, we see a Client that accesses the interface from Listing 8.82. To illustrate that our HelloBean is truly a Singleton Session Bean, we need two clients with almost the same code, Client A is exactly as shown in Listing 8.84. Client B is the same as Client A except that only Client A calls setvalue, Client B does not call setvalue, that is, the following code appears in Client A but not in Client B:

```
proxy.set_value(77);  
  
System.out.println("Just set the value to 77");
```

Since Client B is actually a separate file, it uses a separate proxy object to contain the reference to the HelloBean. So it really is a totally separate client.

If we look at the HelloBean class (see Listing 8.83 again), we see its constructor sets its original value to 999. If a new HelloBean were ever created then its constructor would run again and reset its value to 999.

So now we run Client A first. Its initial value (prior to ClientA calling setvalue) returned is 999. This is the value that was set by the constructor when it was called. Then Client A calls set_value to set the value to 77.

So then we run Client B, which only gets the value (does *not* set the value). Client B returns the value 77. Then every time we

run Client B again after that, even if we run Client B several times, the value returned is always 77.

That is, the HelloBean is *NOT* newly created, because the constructor is never called again.

Therefore, all Clients are using the same Session Bean.

This illustrates that we are using a Singleton Session Bean.

LISTING 8.84 Client A—getvalue Then setvalue to 77

```
package
example.helloworld;
import
javax.naming.InitialC
ontext; import
javax.naming.Context
```

```

try {
    Context ic= new InitialContext();
    Object object_ref =
        ic.lookup("java:global/helloworld/HelloBean");
    // This narrow downcasts the object_ref to type
    HelloWorld, this serves as the proxy HelloWorld
    proxy= (HelloWorld)
        PortableRemoteObject.narrow(object_ref,HelloWorld.class);

    // call the remote method
    on the local proxy String
    returned_string =
    proxy.thehello();
    System.out.println(returned_string);

    int value=proxy.get_value();
    System.out.println("Got original value, the value
    was "+value);

    // Included in Client A only, not
    in Client B proxy.set_value(77);
    System.out.println("Just set the value to 77");
}

```

8.4.5 STATEFUL SESSION BEANS

In this section, we will look at Stateful Session Beans. As you recall from Section 8.4.1, in a Stateful Session Bean, a Client and a Stateful Session Bean are considered to be in a “session” (conversation). When the session (conversation) ends, the state (values of the instance variables) vanishes.

We will first look at a fairly simple Stateful Session Bean example that illustrates the Stateful session/connection oriented behavior. Next, we will look at an example that accesses a Stateful Session Bean asynchronously. Then, we will

expand the server-side code (the Stateful Session Bean) to look at additional server-side processing that is implemented by the container but controlled through the Session Bean: a lifecycle example, and a lifecycle example that forces a passivate.

With a Stateful Session Bean, since its lifetime is the lifetime of the connection or session with the Client, the Stateful Session Bean may potentially need to be instantiated, active, and requiring resources for a considerable amount of time, although sometimes it is not actively being accessed by the Client the entire time.

Sometimes, so many requests come in that the container in the application server can become overcommitted. In this case, the container has to temporarily suspend any entities in the container

that are not currently being actively used, including Stateful Session Beans. In the case of a Stateful Session Bean, this would mean saving all the information about the Stateful Session Bean (this is called *passivating* the Stateful Session Bean) so it can later be retrieved, when it has been accessed again by a remote Client (this is called *activating* the Stateful Session Bean). Note that this means the Stateful Session Bean must be serializable. (Note that only Stateful Session Beans can be activated or passivated, this does not apply to either Stateless Session Beans or Singleton Session Beans.)

8.4.5.1 Stateful Session Bean Example

For our simple Stateful Session Bean example, we will use the same interface previously seen in Listing 8.79 that we used before to illustrate the behavior of a Stateless Session Bean.

In Listing 8.85, we see the Stateful HelloBean that implements the interface in Listing 8.79. This HelloBean is very similar to the Stateless HelloBean we previously saw in Listing 8.78, the biggest difference is that here it is annotated with “@Stateful.”

LISTING 8.85 Stateful HelloBean

```
package
example.helloworld;
import
javax.ejb.Stateful;

@Stateful
public class HelloBean
    implements HelloWorld {
    private int mycount=0;
    public String thehello() {
        return "Hello to all the World!!!!!";
    }
    public int
        getcount() {
            return
                mycount;
        }
    }
```

In Listing 8.86, we see a Client that is very similar to the Client previously shown in Listing 8.81. The difference is that in Listing 8.81, the Client called the “calltheremote” subroutine three times, once for Proxy A, then for Proxy B, then for Proxy C, whereas in this Client in Listing 8.86 calls the “calltheremote” subroutine only twice, once for Proxy A, then for Proxy B.

To clarify a little more: the Client in Listing 8.81 creates two proxy objects: Proxy A and Proxy B. Proxy A does a remote procedure call on the HelloBean, reads the initial value of count, sets its count to 10 and increments it to 11 (this is done in the subroutine on the Client named “calltheremote”).

TABLE 8.2
Output from a Run Illustrating
Session/Connection of Stateful
Bean

	Count Value
Proxy A initial value	0
Proxy A set value	10
Proxy A value incremented to	11
Proxy B initial value	0
Proxy B set value	100
Proxy B value incremented to	101
Final Proxy A value	11
Final Proxy B value	101

Then Proxy B does a remote procedure call on the HelloBean, reads the initial value of count, sets its count to 100 and increments it to 101.

Finally, at the end Proxy A does a getcount (get the value) and Proxy B does a getcount (get the value) on the HelloBean, one after the other.

The output from a run of this Client is shown in Table 8.2.

See how the final counts are different? That's because they are two different proxy objects each of which is handling its own separate session/connection.

Note that the initial value of Proxy A is 0, and the initial value of Proxy B is 0. This is because they are being initialized the first time the Stateful HelloBeans associated with those proxies are created. If we call Proxy A 100 times or 1000, it will always return Proxy A's value and not Proxy B's value, and vice versa. This is because Proxy A has its own session/connection, and Proxy B has its own session/connection.

Conclusion: The Stateful HelloBeans are retaining the state across a session/connection.

LISTING 8.86 Client That Illustrates that Stateful Session
Bean Retains Values Throughout
Session/Connection

```
package  
example.helloworld;  
import  
javax.naming.InitialC  
ontext; import  
javax.naming.Context  
;  
import  
javax.naming.NamingEx
```

```
PortableRemoteObject.narrow(object_ref, HelloWorld.  
  
class); call_the_remote(proxyA, 10); // call all the  
  
methods on the proxy  
  
// Get another different proxy object  
object_ref =  
ic.lookup("java:global/helloworld/HelloBean");  
  
// This narrow downcasts the object_ref to type  
    HelloWorld, this serves as the proxy  
    HelloWorld proxyB= (HelloWorld)  
PortableRemoteObject.narrow(object_ref, HelloWor  
  
ld.class); call_the_remote(proxyB, 100); // call all  
  
the methods on the proxy  
  
System.out.println("Now get the count  
from original proxy"); int  
count_returned = proxyA.getcount();  
System.out.println("proxyA count is  
"+count_returned);  
  
System.out.println("Now get the count  
from the second proxy");  
count_returned = proxyB.getcount();  
System.out.println("proxyB count is  
"+count_returned);  
  
System.out.println("See how the counts are  
different?"); System.out.println("That's because  
they are two different proxy objects");
```

```

        System.out.println("each of which is handling
        its own separate session");
        System.out.println("If I call proxyA 100 times it
        will always return");
        System.out.println("proxyAs value and not
        proxyBs value. And vice versa.");
    }
    catch (NamingException ex) {
        System.out.println("Naming
        exception");
        System.out.println("cause
        is"+ex.getCause());
        System.out.println("stack
        trace is ");
        ex.printStackTrace();
    } catch (Exception ex) {
        System.out.println("gene
        ral exception");
    }
} // end Main()

public static void call_the_remote>HelloWorld proxy,int
value) {
    // call the remote method
    on the local proxy String
    returned_string =
    proxy.thehello();
    System.out.println(returne
    d_string);

    int count_returned =
    proxy.getcount();
    System.out.println("first value of
    count is "+count_returned);

    returned_string =
    proxy.setcount(value);

```

```
System.out.println(returned  
_string);  
count_returned = proxy.getcount();
```

```

System.out.println("count is now
"+count_returned); returned_string
= proxy.incrcount();
System.out.println(returned_string)
;

count_returned = proxy.getcount();

```

8.4.5.2 Stateful Session Bean Asynchronous

First let's look at the difference between *synchronous* and *asynchronous*. In the context of EJBs, *synchronous* means that when a Client calls a Servant/Server, the Client waits until the Server responds before the Client continues operation. Here, *asynchronous* means that the Client calls the Servant/Server and then goes about its business, doing whatever it needs to do, and when the Servant/Server eventually finishes its work and returns the result to the Client, the Client can go ahead and finish up whatever it needed to do related to that call.

An asynchronous interface is shown in Listing 8.87, a Stateful HelloBean that implements this interface is shown in Listing 8.88, and a Client that calls the asynchronous interface is in Listing 8.89.

In EJBs, asynchronous operation is implemented with the `Future<V>` interface, the `AsyncResult<V>` return value, and the `"@Asynchronous"` annotation.

In the asynchronous interface in Listing 8.87, the asynchronous method has the return type:

```
Future<String>
```


LISTING 8.87 HelloWorld Interface—Asynchronous

```
package
example.helloworld;
import
java.util.concurrent.F
uture; import
javax.ejb.Remote;

@Remote
```

In the Stateful HelloBean that implements the asynchronous interface, the asynchronous method is annotated with “@Asynchronous.” It imports `javax.ejb.AsyncResult` and `javax.ejb.Asynchronous`. It defines the asynchronous method as follows:

```
@Asynchronous

public Future<String> thehello()
```

and it asynchronously returns a value as follows:

```
return new AsyncResult<String>("Hello to all the World!!!!");
```

In the asynchronous Client, the asynchronous method is called as follows:

```
Future<String> mystring = proxyA.thehello();
```

LISTING 8.88 Stateful Asynchronous HelloBean

```
package
example.helloworld;
import
javax.ejb.AsyncResult
; import
javax.ejb.Asynchrono
us; import
java.util.concurrent.F
uture;

import
javax.ejb.Statefu
l; import
java.lang.*;

@Stateful
public class HelloBean
    implements HelloWorld {
    private int mycount=0;
    @Asynchronous
    public
        Future<String>
        thehello() { try {
            Thread.sleep (10000); // sleep for 10 seconds
        }
```

To illustrate how an asynchronous call works, the HelloBean pauses in the middle of the call for a considerable period of time:

```
Thread.sleep (10000); // sleep for 10 seconds
```

During the time the HelloBean is taking to process the request, the Client continues to execute.

LISTING 8.89 Asynchronous Client

```
package
example.helloworld;
import
javax.naming.InitialC
ontext; import
javax.naming.Context
;

import
javax.naming.NamingEx
ception; import
javax.rmi.PortableRemote
Object;

import
javax.ejb.AsyncResult;
import
javax.ejb.Asynchronou
s; import
java.util.concurrent.Fu
ture;

public class HelloClient {
    public static void main
        (String[] args) { try {
            Context ic= new InitialContext();

            Object object_ref =
            ic.lookup("java:global/helloworld/HelloBean");

            // This narrow downcasts the object_ref to type
            HelloWorld, this serves as the proxy
            HelloWorld proxyA= (HelloWorld)
                PortableRemoteObject.narrow(object_r
```

```

ef, HelloWorld.class); System.out.println("Now
get the count from original proxy");

Future<String> mystring = proxyA.thehello();

int i=0;
while (!mystring.isDone() )
{
    // We could actually be doing something useful
    now.
    // I'll just count to simulate the
    useful work ☺
    System.out.println("Doing
    useful counting, i = "+i); i++;
}
String myfinalstring =
mystring.get();
System.out.println("string returned
was "+myfinalstring);
}
catch (NamingException ex) {
    // You might get a naming exception if you forgot to
    deploy the jar file
    // You might get a naming exception if you forgot to
    start GlassFish System.out.println("Naming
    exception");
    System.out.println("cause
    is"+ex.getCause());
    System.out.println("stack
    trace is ");
    ex.printStackTrace();
} catch (Exception ex) {
    System.out.println("gene
    ral exception");
}
} // end Main()
} // end class

```

So the results from running the Client look something like this:

```
Now get the count from original proxy
Doing useful counting, i = 0
```

```
Doing useful counting, i = 1
Doing useful counting, i = 2
Doing useful counting, i = 3
Doing useful counting, i = 4
Doing useful counting, i = 5
```

```
... etc. ...
```

```
Doing useful counting, i = 8930
Doing useful counting, i = 8931
Doing useful counting, i = 8932
Doing useful counting, i = 8933
Doing useful counting, i = 8934
```

```
string returned was Hello to all the World!!!!!
```

This illustrates that the Client was able to continue working while the Servant/Server processed the Client's asynchronous request.

8.4.5.3 Stateful Session Bean Lifecycle Callbacks

As previously discussed in Section 8.4.5, with a Stateful Session Bean, since its lifetime is the lifetime of the connection or session with the Client, the Stateful Session Bean may potentially need to be instantiated, active, and requiring resources for a considerable amount of time, although sometimes it is not actively being accessed by the Client the entire time.

Sometimes, so many requests come in for different EJBs (and other entities in the Java EE server managed environment) that the container in the application server can become overwhelmed. In this case, the container has to temporarily suspend any entities in the container that are not currently being actively used, including Stateful Session Beans. In the case of a Stateful Session Bean, this would mean saving all the information about the Stateful Session Bean (this is called *passivating* the Stateful Session Bean) so it can later be retrieved, when it has been accessed again by a remote Client (this is called *activating* the Stateful Session Bean).

Note that this means the Stateful Session Bean must be serializable. (Note that only Stateful Session Beans can be activated or passivated, this does not apply to either Stateless Session Beans or Singleton Session Beans.)

EJB permits methods to be defined that can be invoked at certain points in the lifecycle of an EJB. These are referred to as *callbacks* because from the standpoint of the container in the application server that is what they are (in client/server lingo, a *callback* is when a server calls a routine on a client).

That is, the EJBs start the container running (executing the EJB), so in that situation the EJBs are serving as a client. So when certain events happen on the Servant/Server, the Servant/Server can callback to the Client (call a callback routine) so the Client can handle these events.

A callback can be invoked on any kind of session beans. A Stateless Session Bean, a Singleton Session Bean, and a Stateful Session Bean can all three employ:

- `@PostConstruct`—invoked when the Session Bean is first created
- `@PreDestroy`—invoked just before a Session Bean is destroyed (or removed from the pool of beans)

However, only a Stateful Session Bean can employ:

- `@PostActivate`—invoked after a Stateful Session Bean is reloaded from storage into the container
- `@PrePassivate`—invoked just before a Stateful Session Bean is unloaded from the container and saved away to storage

There is also another annotation that is used to have an instance of a Stateful Bean removed from the container at the end of the call to the annotated method:

- `@Remove`

Routines that are annotated with `@PostConstruct`, `@PreDestroy`, `@PostActivate`, or `@PrePassivate` are invoked by the container in the application server, whereas the Client invokes the method that is annotated by `@Remove`. However, right after the method annotated with “`@Remove`” finishes, the container in the application server will remove the Stateful Session Bean.

In Listing 8.90, we have an interface that we’re going to use to illustrate the lifecycle of a Stateful Session Bean. This interface is similar to interfaces we’ve seen before in Listing 8.79 and in Listing 8.87. However, this one has three new methods:

1. `Initcount`
2. `Finishcounting`
3. `Remove`

LISTING 8.90 Interface to Illustrate Stateful Session Bean Lifecycle

```
package
example.helloworld;
import
javax.ejb.Remote;
@Remote
public interface
    HelloWorld {
        public String
```

The `initcount` routine is meant to be called right after the Session Bean is instantiated. Thus it is annotated with:

- `@PostConstruct`
- `@PostActivate`

We're really ignoring the `PostActivate` in this particular example, because the way we've got it set up, the container in the application server is not busy enough to need to do a `Passivate`. We'll see how that works later in Section 8.4.5.4.

The finishcounting routine is meant to be called just before the Stateful Session Bean is destroyed.

So it is annotated with:

- @PrePassivate
- @PreDestroy

Again, we're really ignoring the PrePassivate in this particular example, because the container in the application server is not busy enough to need to do a Passivate.

The remove routine is meant to be called by the Client to remove the Stateful Session Bean from the container (the routine is called by the Client but the actual removal of the Stateful Session Bean is performed by the container). So it is annotated with:

- @Remove

Note that in this particular HelloBean example, we're using "@Override." @Override forces a compile time check of a method versus the interface. It prevents accidental overloading (instead of intended overriding), by generating a compile-time error when the method signature doesn't exactly match the method signature in the interface. We could have (and probably should have) been using this all along, but to save confusion and a necessary explanation I left it off until now.

LISTING 8.91 HelloBean to Illustrate Stateful Session Bean Lifecycle

```
package
example.helloworld;
import
javax.ejb.Stateful;
import
javax.annotation.PostCon
struct; import
javax.annotation.PreDest
roy; import
javax.ejb.Remove;
import
javax.ejb.PostActi
vate;      import
javax.ejb.PrePassi
vate; @Stateful
public class HelloBean
    implements HelloWorld {
    private int mycount=0;
```

```

        mycount=value;
        return "mycount has been set" ;
    }
    @Override
    public String
        incrcount() {
        mycount++;
        return "mycount has been incremented" ;
    }
    @PrePassivate
    @PreDestroy
    @Override
    public void
        finishcounting() {
        mycount=7777;
        System.out.println("Doing PreDestroy or

```

We use the Client in Listing 8.92 to call the interface from Listing 8.90. We have two separate proxy objects, Proxy A and Proxy B. We find the interface using JNDI for each of these, then we call `getcount()` separately on each proxy object. Finally, we call `remove()` on each proxy object.

When we run the Client, the output looks like this:

```

Now get the count from Proxy A
proxyA count is 9999

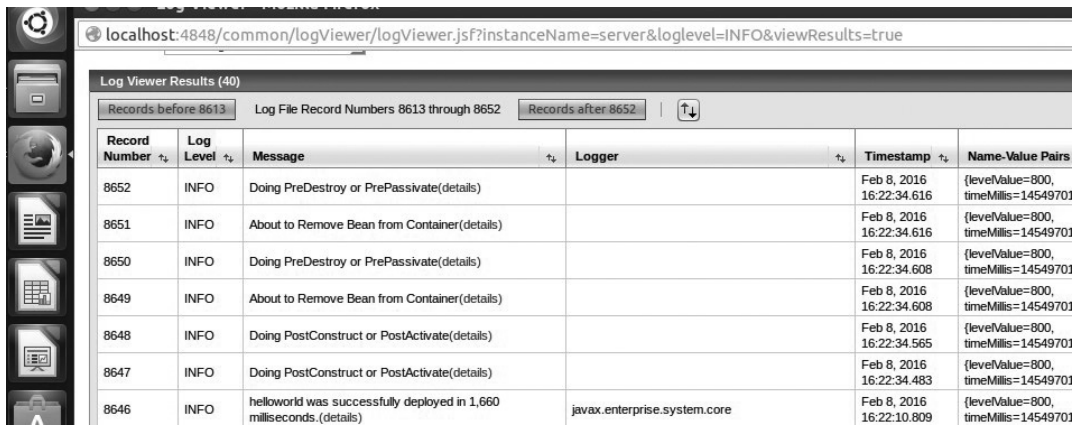
```

```

Now get the count from Proxy B
proxyB count is 9999

```

However, for this example, the interesting results are shown in the Server log file. A screenshot of the server log file is available in Figure 8.10.



localhost:4848/common/logViewer/logViewer.jsf?instanceName=server&loglevel=INFO&viewResults=true

Log Viewer Results (40)

Records before 8613 | Log File Record Numbers 8613 through 8652 | Records after 8652 | [↑↓]

Record Number ↑	Log Level ↑	Message ↑	Logger ↑	Timestamp ↑	Name-Value Pairs
8652	INFO	Doing PreDestroy or PrePassivate(details)		Feb 8, 2016 16:22:34.616	{levelValue=800, timeMillis=14549701}
8651	INFO	About to Remove Bean from Container(details)		Feb 8, 2016 16:22:34.616	{levelValue=800, timeMillis=14549701}
8650	INFO	Doing PreDestroy or PrePassivate(details)		Feb 8, 2016 16:22:34.608	{levelValue=800, timeMillis=14549701}
8649	INFO	About to Remove Bean from Container(details)		Feb 8, 2016 16:22:34.608	{levelValue=800, timeMillis=14549701}
8648	INFO	Doing PostConstruct or PostActivate(details)		Feb 8, 2016 16:22:34.565	{levelValue=800, timeMillis=14549701}
8647	INFO	Doing PostConstruct or PostActivate(details)		Feb 8, 2016 16:22:34.483	{levelValue=800, timeMillis=14549701}
8646	INFO	helloworld was successfully deployed in 1,660 milliseconds.(details)	javax.enterprise.system.core	Feb 8, 2016 16:22:10.809	{levelValue=800, timeMillis=14549701}

FIGURE 8.10 Screenshot of server log file from GlassFish—PostConstruct and PreDestroy.

LISTING 8.92 Client to Illustrate Stateful Session Bean Lifecycle

```
package
example.helloworld;
import
javax.naming.InitialC
ontext; import
javax.naming.Context
;
import
javax.naming.NamingEx
ception; import
javax.rmi.PortableRemote
Object;

public class HelloClient {
    public static void main
        (String[] args) { try {
        Context ic= new InitialContext();

        // Get a proxy object
        Object object_ref =
        ic.lookup("java:global/helloworld/HelloB
        ean"); HelloWorld proxyA= (HelloWorld)
        PortableRemoteObject.narrow(object_ref,HelloWorl
        d.class);

        System.out.println("Now get the
        count from ProxyA"); int
        count_returned =
        proxyA.getcount();
        System.out.println("proxyA count
        is "+count_returned);
```

```

// Get another proxy object
object_ref =
ic.lookup("java:global/helloworld/HelloB
ean"); HelloWorld proxyB= (HelloWorld)
PortableRemoteObject.narrow(object_re
f,HelloWorld.class);

System.out.println("Now get the
count from ProxyB");
count_returned =
proxyB.getcount();
System.out.println("proxyB count is
"+count_returned);

// Explicitly remove the beans
from the container
proxyA.remove();
proxyB.remove();

}
catch (NamingException ex) {
    System.out.println("Naming
exception");
    System.out.println("cause
is"+ex.getCause());
    System.out.println("stack
trace is ");
    ex.printStackTrace();
} catch (Exception ex) {
    System.out.println("gene
ral exception");
}

} // end Main()

} // end class

```

Figure 8.10, the server log, must be read from bottom to top. Thus, the first event that was logged was that HelloWorld was deployed. The next event logged was “Doing PostConstruct or PostActivate.” If you look back at Listing 8.91, this was the `initcount()` routine running. It set the counter to 9999. This is called twice, once for Proxy A on the Client, and once for Proxy B.

Note that the calls to `getcount()` do not result in any messages being logged to the server log, there just weren’t any put into this method (we would have had to do a `System.out.println` using some kind of interesting string, and we just didn’t happen to do that.)

Next, we see from the logged messages (“About to Remove Bean from Container”) that Remove is called for Proxy A, which then (when the `remove()` method completes) results in a call from the container to `finishcounting()`, resulting in the message “Doing PreDestroy or PrePassivate” in the log. This is followed immediately by the container removing the Stateful Session Bean from the container. Then this repeats for Proxy B.

8.4.5.4 Stateful Session Bean Lifecycle Callbacks Forces Passivate

This example is another example illustrating the lifecycle of a Stateful Session Bean. However, in this case we’re going to force a Passivate so we can see what that looks like. We will use the interface in Listing 8.93 to do this. This interface is similar to the interface we previously saw in Listing 8.90. However, here we have split the `initcount()` routine into two separate routines, `initcount_PostConstruct()` and `initcount_PostActivate()` so we can distinguish between creation of the Stateful Session Bean and re-activation of an existing Stateful Session Bean that was previously Passivated. Similarly, we have split the `finishcounting()` routine into two separate routines, `finishcounting_PrePassivate()` and `finishcounting_PreDestroy()` so we can distinguish between a Stateful Session Bean that is about to be destroyed, and a

Stateful Session Bean that is about to be Passivated.

You should remember from our previous discussions in Sections 8.4.5 and 8.4.5.3 that so many requests come in for different EJBs (and other entities in the Java EE server managed environment) that the container in the application server can become overcommitted. In this case, the container has to temporarily suspend any entities in the container that are not currently being actively used, including Stateful Session Beans. In the case of a Stateful Session Bean, this would mean saving all the information about the Stateful Session Bean (this is called *passivating* the Stateful Session Bean) so it can later be retrieved, when it has been accessed again by a remote Client (this is called *activating* the Stateful Session Bean). Note that this means the Stateful Session Bean must be serializable. (Note that only Stateful Session Beans can be activated or passivated, this does not apply to either Stateless Session Beans or Singleton Session Beans.)

LISTING 8.93 Interface to Illustrate Stateful
Session Bean Lifecycle with Forced
Passivate

```
package  
example.helloworld;  
import  
javax.ejb.Remote;  
@Remote
```

```
public String incrcount();  
public void  
finishcounting_PrePassiva  
te(); public void
```

If you look at the implementation of `initcount_PostConstruct()` and `initcount_PostActivate()` in Listing 8.94, you will see that `initcount_PostConstruct()` is annotated with “`@PostConstruct`” and `initcount_PostActivate()` is annotated with “`@PostActivate`.” Similarly, the method

`finishcounting_PrePassivate()` is annotated with “`@PrePassivate`” and the method `finishcounting_PreDestroy()` is annotated with “`@PreDestroy`.”

LISTING 8.94 HelloBean to Illustrate Stateful Session Bean Lifecycle with Forced Passivate

```
package
example.helloworld;
import
javax.ejb.Stateful;
import
javax.annotation.PostCon
struct; import
javax.annotation.PreDest
roy; import
javax.ejb.Remove;
import
javax.ejb.PostActi
vate;      import
javax.ejb.PrePassi
vate; @Stateful
public class HelloBean
    implements HelloWorld {
    private int mycount=0;

    @Override
    public String thehello() {
        return "Hello to all the World!!!!!!";
```

```

@Override
public String
    setcount(int
        value) {
    mycount=value
    ;
    return "mycount has been set" ;
}
@Override
public String incrcount() {
    mycount++;
    return "mycount has been incremented" ;
}
@PrePassivate
@Override
public void
    finishcounting_PrePassivate() { mycount=7777;
    System.out.println("Doing PrePassivate");
}

```

Now deploy the Bean shown in Listing 8.94; your server log will look as shown in the screenshot in Figure 8.11.

Okay, this is fairly straightforward so far. Now comes the harder part of this project. We want to use up resources *inside* the container, so the container is forced to passivate a Stateful Session Bean. To do this, I have chosen to use a servlet. I'm using this in advance of Chapter 11, where I describe servlets in much more detail. However, it's very convenient to use one for this particular example, as you will see. I'll try to explain everything you really need to know for this particular example here.

The big advantage to using a servlet is that the servlet itself is running *inside* the container in the application server. So if

I use up resources in the servlet, I'm using up resources *inside* the container which is what I want when I'm trying to force the container to passivate some Stateful Session Beans.

Log Viewer Results (40)							
Records before 1808		Log File Record Numbers 1808 through 1847			Records after 1847		⬆⬇⬆
Record Number	Log Level	Message	Logger	Timestamp	Name-Value Pairs		
1847	INFO	helloworld was successfully deployed in 2,902 milliseconds.(details)	javax.enterprise.system.core	Feb 29, 2016 20:52:23.188	{levelValue=800, timeMillis=1456800743188}		
1846	INFO	Loading application [helloworld] at [helloworld](details)	javax.enterprise.web	Feb 29, 2016 20:52:23.072	{levelValue=800, timeMillis=1456800743072}		

FIGURE 8.11 Screenshot of server log file from GlassFish—Initial deployment.

A servlet runs in a container in an application server. Most servlets extend the `HttpServlet` class.

Thus, inputs come to the servlet in the form of HTTP messages (from the HTTP protocol).

A servlet that extends the `HttpServlet` class can handle GET, POST, PUT, and DELETE messages. The `HttpServlet` method `doGet` would handle GET messages, `doPut` handles PUT messages, `doPost` handles POST messages, and `doDelete` handles DELETE messages. These correspond to Create-Read-Update-and-Delete (CRUD) basic operations for managing persistent storage: PUT/`doPut` and POST/`doPost` correspond to Create, GET/`doGet` corresponds to Read, POST/`doPost` and PUT/`doPut` correspond to update, and DELETE/`doDelete` corresponds to Delete.

A servlet can be used to implement RESTful web services. Note that just using a servlet doesn't make a web service RESTful, it has to also be controlling a resource (we'll talk a lot more about this later in Chapter 11). But since a servlet implements the CRUD operations that would be used on persistent storage (an example of a resource), this is usually easy to do. However, servlets have often been considered to be the *assembly language* of RESTful middleware, that is, they're very low level compared to some other RESTful middleware options.

Yes, instead of using a servlet I could have done this by instantiating various EJBs using outside POJO clients (probably better off using Stateful Session Beans since they can hang around a while, Stateless Beans we'd just be using one of a pool and using Singletons would have required having a whole bunch of different Singletons—think about it, exercise for the student ☺). But using a servlet has a couple of advantages. First, using one here allows me to briefly introduce what a servlet is and does, which is yet another useful technology that you the student really want to learn. ☺

Also, the problem with using POJO Clients to call Stateful Session Beans is that even with Stateful Session Beans I would need to fix a way for the Stateful Session Beans to hang around and not finish—big wait loops/sleep, etc. Also, a servlet has a convenient thing called an *attribute*, that makes it easy to store resources inside the servlet (and thus inside the container). (Attributes can also be used to pass information between servlets.) So I'm making use of this here. What I'm going to do in the servlet is I will create lots of Proxy objects that reference a Stateful Session Bean. Then I will store all those proxies in my servlet attribute, in session scope (we'll talk about that in a second), to use up container resources. Attributes and parameters in a Java EE web application are related to the *servlet scope* that specifies where that attribute or parameter is available for use. The scope of the servlet is how the servlet maintains a state.

To use attribute information within a scope, these are the routines that one could call:

- SetAttribute(name, object) saves the object along with its name.
- GetAttribute(name) returns the object associated with the specified name
- RemoveAttribute(name) removes the object from the context
- GetAttributeNames() returns an enum containing the names of all attributes
 - Could do:
 - Enumeration

`the_names=request.getAttributeNames();`

Servlets have three scopes:

1. Application (Context) scope—Lifetime of the Web Application. Available while a web application is in service, from start to finish (or reload). Valid during interaction of all users with a particular web application.
 - javax.servlet.ServletContext interface—defines methods used by a servlet to communicate with the servlet container. There is one context for each web application in a JVM.
 - To access this, you can do:
 - ServletContext mycontext = getServletContext();
 - Or:
 - ServletConfig myconfig = getServletConfig();
 - ServletContext
 - mycontext=myconfig.getServletContext();

2. Request scope—Duration of the Request. From when an HTTP request is received by the servlet until an HTTP response is generated by the servlet. Valid during a single user's interaction with a web application during a single HTTP request.
 - `Javax.servlet.http.HttpServletRequest` interface—provides HTTP requests to HTTP servlets.
 - An `HttpServletRequest` object is created by the servlet container and passed to `doGet`, `doPut`, etc. in the servlet
 - To access this scope, you can call `getAttribute(name)`, etc. on the request object passed into `doGet` (from inside `doGet()`), etc.
3. Session—Lifetime of the Client. From when a client establishes a connection with the application until the connection is closed. Valid across multiple HTTP requests from a single user.
 - `Javax.servlet.http.HttpSession` interface—used to create a session from an HTTP client (one user) to an HTTP server that persists for a specified period of time.
 - `HttpSession session = request.getSession();`
 - `getSession()`—gets the session, if there isn't already an `HttpSession` object, it creates one.
 - If cookies are enabled:
 - It generates a `JSESSIONID` cookie and value that is used to identify this session when further requests arrive from this client. This `JSESSIONID` cookie is sent to the Client in a response, the Client uses it in future requests.
 - The HTTP header, if displayed, would include:
 - Set-Cookie: `JSESSIONID=..long messy hex value...`; Path=/example; Expires= ..date..; Secure
 - (the secure attribute means that cookies should only be used in
 - If cookies are not enabled, it uses URL rewriting:
 - A `JSESSIONID` is added at the end of the URL, and the request is redirected to this URL
 - `www.mstuff.com/hello` becomes `www.mystuff.com/hello/jsessionid=...long messy hex value...`

URL rewriting consists of telling the Server to redirect a request from one URL to a different URL. Typically URL rewriting is done so a more readable URL can be the one the Client accesses first, then that URL is redirected to a less readable internal URL (the domain name is the same in both cases, but the rest of the URL would need to be redirected). In this particular case, however, URL rewriting is used to establish a session, so the more readable URL is rerouted to a

less readable URL.

In this particular example in Listing 8.95 we are creating a number of proxy objects that reference our interface (from Listing 8.93). We do this by:

```
ic.lookup("java:global/helloworld/HelloBean");
```

Similar to what we saw earlier in Section 8.4.3.1, we are using the *java:global* namespace here.

Names in the java:global JNDI namespace are accessible to any application.

Then, again similar to what we saw earlier in Section 8.4.3.1, we must create an appropriate proxy object to represent the HelloWorld interface in the Client. We do this via a downcast from the object reference returned from JNDI:

```
example.helloworld.HelloWorld proxyA=  
(example.helloworld.HelloWorld)  
PortableRemoteObject.narrow(object_ref,example.helloworld.HelloWorld.class);
```

We're storing these proxy objects in an attribute as follows:

```
request.getSession().setAttribute("current_bean"+i,proxyA);
```

This is storing the bean in Session scope. After we store a sufficient quantity of beans to use up the GlassFish resources allocated for the storage of session beans, we will force a passivate, and we will be able to see the passivate in a screenshot of the server log, similar to what we saw previously in Section 8.4.5.3 in regard to `PostConstruct` and `PreDestroy`.

LISTING 8.95 A Servlet to Force a Passivate

```
package
example.helloworld; import
java.io.*;
import
javax.servlet.annotation.We
bServlet; import
javax.servlet.*;
import javax.servlet.http.*;

import
javax.naming.InitialCo
ntext; import
javax.naming.Context;
import
javax.naming.NamingEx
ception; import
javax.rmi.PortableRemote
Object;
```

```

    @WebServlet (
        urlPatterns={"/myservlet"}
    )
    public class myservlet extends HttpServlet {

        public void
            doGet(HttpServletRequest request,
                HttpServletResponse response)
                throws ServletException, IOException
        {
            // put a text/html header in
            // the HTTP response
            response.setContentType("text/html");

            // Following output will show up in GlassFish log file
            // because System.out is standard output it will show
            // up on the console
            System.out.println("Inside myservlet in doGet
            method-trying");

            // Write a small simple web page
            // into the HTTP response
            PrintWriter myout =
                response.getWriter();
            myout.println("inside the servlet");

            example.helloworld.HelloBean mybean;
            String
                original_bean_counter;

            original_bean_counter =

                request.getParameter("thecount");

```

```
int bean_counter;
```

```
System.out.println("o b c is
"+original_bean_counter);
bean_counter=Integer.parseInt(original_bean_counter)
;

int

i;

try

{

    i=0;
    while (i < bean_counter)
    {
        Context ic= new InitialContext();
        // Keep acquiring proxy objects to point to
        HelloBean
        Object object_ref =
        ic.lookup("java:global/helloworld/HelloBean")
        ; example.helloworld.HelloWorld proxyA=
        (example.helloworld.
        HelloWorld)
        PortableRemoteObject.narrow(object_ref,e
        xample.helloworld.
        HelloWorld.class);

        // An attribute in a servlet is an object that can
        be stored
        // in request scope, session scope or
        application scope.
        // Here the object is being stored in session
        scope.
```

```

        // Attributes can be used to pass information
        between
    // servlets.
    // setAttribute(name, object) saves the object
    along with its name.
    // getAttribute(name) returns the object
    associated with the
    // specified name, while
    removeAttribute(name) removes the
    // object from the context.
    request.getSession().setAttribute("cur
    rent_bean"+i,proxyA); i++;
    }
}

catch (NamingException ex) {
    System.out.println("Naming
    exception");
    System.out.println("cause
    is"+ex.getCause());
    System.out.println("stack
    trace is ");
    ex.printStackTrace();
} catch (Exception ex) {
    System.out.println("general exception");
}
}
}

```

We read the number of proxy objects to store from a parameter on an HTTP GET that is sent to the servlet. This can be done through the browser as follows:

- <http://localhost:8080/helloworld/helloworld?thecount=515>

2390	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.306	{levelValue=800, timeMillis=1456801564306}
2389	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.300	{levelValue=800, timeMillis=1456801564300}
2388	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.299	{levelValue=800, timeMillis=1456801564299}
2387	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.293	{levelValue=800, timeMillis=1456801564293}
2386	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.288	{levelValue=800, timeMillis=1456801564288}
2385	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.284	{levelValue=800, timeMillis=1456801564284}
2384	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.281	{levelValue=800, timeMillis=1456801564281}
2383	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.273	{levelValue=800, timeMillis=1456801564273}
2382	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.269	{levelValue=800, timeMillis=1456801564269}
2381	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.267	{levelValue=800, timeMillis=1456801564267}
2380	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.264	{levelValue=800, timeMillis=1456801564264}
2379	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.260	{levelValue=800, timeMillis=1456801564260}
2378	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.253	{levelValue=800, timeMillis=1456801564253}
2377	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.249	{levelValue=800, timeMillis=1456801564249}
2376	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.247	{levelValue=800, timeMillis=1456801564247}
2375	INFO	Doing PrePassivate(details)	Feb 29, 2016 21:06:04.187	{levelValue=800, timeMillis=1456801564187}
2374	INFO	Doing PostConstruct(details)	Feb 29, 2016 21:06:04.162	{levelValue=800, timeMillis=1456801564162}
2373	INFO	Doing PostConstruct(details)	Feb 29, 2016 21:06:04.161	{levelValue=800, timeMillis=1456801564161}

FIGURE 8.12 Screenshot of server log file from GlassFish—PrePassivate.

Or it can be sent with a cURL command as follows:

- `curl --request GET "http://localhost:8080/helloworld/myservlet?thecount=515"`

The variable “thecount” is read as follows:

```
original_bean_counter = request.getParameter("thecount");
```

```
int bean_counter;  
bean_counter=Integer.parseInt(original_bean_counter);
```

where the `getParameter("thecount")` acquires the parameter from the GET request, and the `parseInt` converts it to an integer counter.

We picked 515 here because the GlassFish parameter `max-cache-size` defaults to 512. The `max-cache-size` parameter is a hint to GlassFish as to how many session beans GlassFish should support.

After we send the GET to the servlet, we can check the server log file to see that the passivate has occurred. (If for some reason you don't see a passivate in your server log file, just increase the size of the count.)

An example of what you're looking for is shown in the screenshot in Figure 8.12.

CORBA EXERCISES

1. In CORBA, define the IDL required to send an array of structs containing student data from the Client to the Server, where the struct contains a string called “name” and a float called “GPA.” The array should be 100 items long.
2. In CORBA, you know that String_var is a wrapper around char *. How might you extract the char * contents out of a String_var wrapper?
3. In CORBA:

Given:

```
interface A
{
```

```
    ...
```

```
};
```

```
interface B: A {
```

```
    ...
```

```
};
```



```
A_ptr aptr = ... ; // Initialize
B_ptr bptr = B::_narrow(aptr);
```

What does the `_narrow()` routine do? (Give a short but complete explanation)

4. Does CORBA IDL support an integer data type? Why?
5. Does CORBA idl support multiple inheritance of interfaces?
6. Given:

```
myecho * the_echo = new myecho;
echo_var echo_obj = myecho._this();
```

What is the reason for the `_this()` routine in the above code?

7. Does an IDL attribute contain any data storage? Why or why not?
8. Create a variable of type “Any” and store “long” data in that variable, using C++ CORBA. Then do it again using Java CORBA.
9. In CORBA using Java, how do you access an IDL constant?
10. Why is it acceptable to use a “/” as part of a valid name in the CORBA Naming Service?
11. Is a naming graph in the CORBA Naming Service allowed to have loops?
12. How does the `corbaloc:rir` format work?
13. Does CORBA use HTTP?
14. If you want all incoming CORBA requests to be handled by the same servant, how would you set up the CORBA server?
15. You wish to write a function to update, or change your string. Your string is defined as follows in C++/CORBA on the client side:


```
CORBA::String_var msg1= CORBA::string_dup("Hi there");
```

You will call `myfunc` as follows:

```
myfunc(msg1);
```

What is the correct way to define the parameter definition in the servant `myfunc`?

.NET REMOTING EXERCISES

1. When doing Marshal by Value in .NET Remoting, what is the primary requirement for the remote object being called? How does the process of calling this remote object work?
2. If you're not going through a firewall, and you're always using the same platform, which .NET format-ter should you use and why?
3. What happens in .NET Remoting, when using Marshal by Value, if a serialized class has some member that is not serializable?
4. What kind of .NET Remoting channel would you use if you're going through firewalls?
5. What .NET Remoting activation mode would you use if you want one servant to handle all clients?
6. What .NET Remoting activation mode would you use if you're concerned about scalability?
7. Is using one servant to handle all clients good when you're considering scalability?
8. How many channels can you register per port number?
9. How many channels can you register per appdomain?
10. When using asynchronous .NET Remoting with a callback function, are there any restrictions as to where the call to `EndInvoke` should be located?
11. If you wanted more than one result to be returned to a callback function, how would you do this?
12. What does the `[OneWayAttribute]` mean?

13. What happens when a lease expires?
14. What are the limitations with using client-activated objects?
15. When should you use a named pipe?

EJB EXERCISES

1. Can you simply say “Java Bean” to mean “EJB”?
2. Is it possible to have one EJB that handles many (maybe all) clients? If so, what kind?

3. What kind of EJB would you use if you are primarily worried about scalability?
4. When would you use a Stateful Session Bean?
5. What possible methods can a client use to access an EJB?
6. In what situation can a client be used to call a local interface?
7. What is meant by “passivating” an EJB?
8. Can a stateless session bean be passivated? Why?
9. Can a singleton session bean be passivated? Why?
10. What are the three contexts (scopes) of a Java servlet? Explain.
11. How does a Java servlet establish and maintain a session? And how is this different from how an EJB maintains a session?
12. How can you tell how many session beans GlassFish will support?
13. How does an asynchronous client retrieve its result when the asynchronous call has finished?
14. What is the purpose of the “@Startup” annotation?
15. Why do you have to do an explicit call to narrow() in the following code ?

```
Context ic= new InitialContext();

Object object_ref = ic.lookup("java:global/helloworld/HelloBean");

// This narrow downcasts the object_ref to type HelloWorld, this serves as the
proxy HelloWorld proxy= (HelloWorld)

PortableRemoteObject.narrow(object_ref,HelloWorld.class);
```

CONCEPTUAL QUESTIONS

1. Suppose you want your distributed transactions to be stateless? How would you do this in CORBA? How does this compare to how you would do the same with .NET Remoting or EJBs? What are the pros and cons? Now repeat for stateful transactions?
2. Which one of the technologies we examined is best suited to working on different operating systems? Why? Is this always a good thing? What about supporting different languages?
3. Think about persistent objects, such as objects that provide a front end to a database. What are the pros and cons for CORBA, .NET Remoting, and EJBs for handling persistent objects?
4. Which of these technologies is the hardest to learn? Justify your answer with examples. Which of these technologies is the hardest to use (assuming you have already spent the time to learn it). Justify your answer with examples.
5. Design your own object-oriented/component-based middleware. ☺ Which pieces would you pick from each of these three technologies: CORBA, .NET Remoting, EJBs? Which would you definitely throw out? Write a brief description of how your middleware would work, and justify why it's better than CORBA, .NET Remoting, or EJBs.

BIBLIOGRAPHY

CDC 6600, https://en.wikipedia.org/wiki/CDC_6600 (accessed January 9, 2016).

CDC Cyber Computers, https://en.wikipedia.org/wiki/CDC_Cyber (accessed January 9, 2016).

Erlang. Corba Assisted and User-defined Exceptions. 1997. http://www.erlang.org/doc/apps/orber/ch_exceptions.html (accessed January 9, 2016).

- Esangbedo, I.S. Reflection in C# Tutorial. CodeProject. 2007. <http://www.codeproject.com/Articles/17269/Reflection-in-C-Tutorial> (accessed March 24, 2016).
- Etzkorn, L., Sherrill, J., and O'Guin, R. The CORBA Notification Service: Applicability in a Real Time, Embedded Environment, *Embedded Systems Programming* (renamed Embedded Systems Design), 2002. <http://www.embedded.com/print/4024457> (accessed March 24, 2017).
- Giddings, V., and Beckwith, B. Real-Time CORBA Tutorial. OMG Workshop for Distributed Object Computing for Real Time and Embedded Systems. Arlington, VA, USA. 2003. http://www.omg.org/news/meetings/workshops/RT_2003_Manual/Tutorials/T1_RTCORBA_Giddings.pdf (accessed January 31, 2016).
- Gupta, S. A Performance Comparison of Windows Communication Foundation (WCF) with Existing Distributed Communication Technologies. MSDN. 2007. <https://msdn.microsoft.com/en-us/library/bb310550.aspx> (accessed March 18, 2016).

- Heartin. Application, Request, Session, and Page Scopes in Servlets and JSPs. 2013. <http://www.javajee.com/application-request-session-and-page-scopes-in-servlets-and-jsps> (accessed February 10, 2016).
- Henning, M. *The Rise and Fall of CORBA*. Vol. 4, Issue 5, 2006, <https://queue.acm.org/detail.cfm?id=1142044> (accessed November 28, 2015).
- Henning, M., and Vinoski, S. *Advanced CORBA Programming with C++*. 1999. Addison-Wesley, Reading, MA.
- Ironsides, E., Etzkorn, L., and Zajac, D., Examining CORBA Interoperability, *Dr. Dobbs[®] Journal*, June, 2001, 111–122.
- Java Community Process. JSR 153: Enterprise JavaBeans 2.1. 2003. <https://jcp.org/en/jsr/detail?id=153> (accessed February 7, 2016).
- Java EE 6 Tutorial. Asynchronous Method Invocation. 2013. <http://docs.oracle.com/javaee/6/tutorial/doc/gkkqg.html> (accessed February 8, 2016).
- Java EE 6 Tutorial. The Lifecycle of Enterprise Beans. 2013. <http://docs.oracle.com/javaee/6/tutorial/doc/giplj.html> (accessed February 9, 2016).
- Java EE 6 Tutorial. Using Scopes. 2013. <http://docs.oracle.com/javaee/6/tutorial/doc/gjbbk.html> (accessed February 9, 2016).
- JavaHowTo. Servlet without web.xml. 2009. <http://javahowto.blogspot.com/2009/10/servlet-without-webxml.html> (accessed February 7, 2016).
- JavaTpoint. Attribute in Servlet. 2016. <http://www.javatpoint.com/attribute> (accessed February 7, 2016).
- Johnson, M. A Beginner's Guide to EJBs. Java World. 1998. <http://www.javaworld.com/article/2076777/learn-java/a-beginner-s-guide-to-enterprise-javabeans.html?page=2> (accessed March 26, 2016).
- Kobashi Computing. How to Force a Windows Application to be Run with Administrator Privileges. 2013. <http://www.kobashicomputing.com/how-to-force-a-windows-application-to-be-run-with-administrator-privileges> (accessed March 26, 2016).
- Kozierok, C.M. IP Multicasting. TCP/IP Guide. 2005. http://www.tcpipguide.com/free/t_IPMulticasting.htm (accessed January 22, 2016).
- Maravitsas, N. EJB Passivation and Activation Example. 2013. www.javacodegeeks.com/2013/08/ejb-passivation-and-activation-example.html (accessed February 7, 2016).
- McHale, C. Chapter 6 POA Policies. CORBA Explained Simply. www.ciaranmchale.com/corba-explained-simply/poa-policies.html (accessed January 24, 2016).
- McHale, C. Chapter 10 Interoperable Object Reference. CORBA Explained Simply. <http://www.ciaranmchale.com/corba-explained-simply/interoperable-object-reference.html> (accessed January 19, 2016).
- McHale, C. Chapter 12 The Corbaloc and Corbaname URLs. <http://www.ciaranmchale.com/corba-explained-simply/the-corbaloc-and-corbaname-urls.html#toc63> (accessed January 19, 2016).
- Mehra, P. Remote Object Lifetime in .NET. C#Corner. 2010. <http://www.c-sharpcorner.com/UploadFile/puranindia/remote-object-lifetime-in-net/> (accessed March 26, 2016).
- MSDN. Remotable Objects. 2003. [https://msdn.microsoft.com/en-us/library/aa720494\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa720494(v=vs.71).aspx) (accessed March 24, 2016).

MSDN. Lifetime Leases. 2006. [https://msdn.microsoft.com/en-us/library/23bk23zc\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/23bk23zc(v=vs.85).aspx) (accessed March 26, 2016).

MSDN. Remoting Example: Asynchronous Remoting. 2006. [https://msdn.microsoft.com/en-us/library/Osa925ka\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/Osa925ka(v=vs.85).aspx) (accessed March 26, 2016).

MSDN. .NET Framework Remoting Overview. 2010. [https://msdn.microsoft.com/en-us/library/kwdt6w2k\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/kwdt6w2k(v=vs.100).aspx) (accessed March 18, 2016).

MSDN. Application Manifests. 2014. [https://msdn.microsoft.com/en-us/library/aa374191\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa374191(v=vs.85).aspx) (accessed March 24, 2016).

MSDN. 2.2.6 Lease State. [MS-NRSL]: .NET Remoting: Lifetime Services Extension. 2015. <https://msdn.microsoft.com/en-us/library/cc236920.aspx> (accessed March 26, 2016).

MSDN. Assembly Manifest. 2015. [https://msdn.microsoft.com/en-us/library/1w45z383\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/1w45z383(v=vs.110).aspx) (accessed March 24, 2016).

MSDN. Blocking Application Execution Using an AsyncWaitHandle. 2015. [https://msdn.microsoft.com/en-us/library/ms228962\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms228962(v=vs.110).aspx) (accessed March 26, 2016).

MSDN. Calling Synchronous Methods Asynchronously. 2015. [https://msdn.microsoft.com/en-us/library/2e08f6yc\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/2e08f6yc(v=vs.110).aspx) (accessed March 26, 2016).

MSDN. Manifest Generation in Visual Studio 2015. 2015. <https://msdn.microsoft.com/en-us/library/ms235229.aspx> (accessed March 24, 2016).

MSDN. System Reflection Namespace. 2015. [https://msdn.microsoft.com/en-us/library/system.reflection\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.reflection(v=vs.110).aspx) (accessed March 24, 2016).

MSDN. Application Domains. 2016. [https://msdn.microsoft.com/en-us/library/2bh4z9hs\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/2bh4z9hs(v=vs.110).aspx) (accessed March 18, 2016).

Object Management Group. CORBA. <http://www.corba.org/> (accessed March 24, 2017). Object Management Group. Common Object Request Broker Architecture (CORBA) Specification, Version 3.3

Part 1: CORBA Interfaces. 2012. <http://www.omg.org/spec/CORBA/3.3/Interfaces/PDF/> (accessed January 22, 2016).

Object Management Group. History of CORBA. 2015. http://www.omg.org/gettingstarted/history_of_corba.htm (accessed November 28, 2015).

OmniORB. Chapter 12 The Dynamic Skeleton Interface. <http://www.cl.cam.ac.uk/research/dtg/attarchive/omniORB/doc/3.0/omniORB/omniORB012.html> (accessed January 31, 2016).

OmniORB3. Chapter 4 Interoperable Naming Service. <http://www.cl.cam.ac.uk/research/dtg/attarchive/omniORB/doc/3.0/omniORB/omniORB004.html> (accessed January 19, 2016).

OmniORB4. Chapter 6 Interoperable Naming Service. <http://omniorb.sourceforge.net/omni40/omniORB/omniORB006.html> (accessed January 19, 2016).

OmniORB4 The omniNames CORBA Naming Service. 2014. <http://omniorb.sourceforge.net/omni42/omniNames.html> (accessed January 22, 2016).

Oracle. Chapter 22 Enterprise Beans. The Java EE 6 Tutorial. 2013. <https://docs.oracle.com/javaee/6/tutorial/doc/gijsz.html> (accessed February 7, 2016).

Oracle. CORBA Technology and the Java Platform Standard Edition. <http://docs.oracle.com/javase/7/docs/technotes/guides/idl/corba.html> (accessed November 28, 2015).

Oracle. max-cache-size. GlassFish Server 3.1 Application Deployment Guide. 2010. https://docs.oracle.com/cd/E18930_01/html/821-2417/beauo.html (accessed February 29, 2016).

Oracle. Oracle8i Enterprise JavaBeans and CORBA Developer's Guide, Chapter 3, https://docs.oracle.com/cd/A81042_01/DOC/java.816/a81356/corba.htm (accessed November 28, 2015).

Oracle. Serializable Objects. <http://docs.oracle.com/javase/jndi/tutorial/objects/storing/serial.html> (accessed January 31, 2016).

Oracle. Simple EJBs 3.0 Examples. 2016. https://docs.oracle.com/cd/E13222_01/wls/docs100/ejb30/examples.html (accessed February 7, 2016).

Oracle. Using the Dynamic Invocation Interface. https://docs.oracle.com/cd/E13203_01/tuxedo/tux91/creclient/dii.htm#1013394 (accessed January 31, 2016).

Oracle. The @Remove Method. The Java EE 6 Tutorial. 2010. <https://docs.oracle.com/cd/E19798-01/821-1841/bnboi/index.html> (accessed February 9, 2016).

Oracle. What Is a Session Bean. 2013. <http://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html> (accessed February 2, 2016).

PCReview. .NET Remoting TCP or HTTP? 2005. <http://www.pcreview.co.uk/threads/net-remoting-tcp-or-http.1234716/> (accessed March 25, 2016).

Programmers Stack Exchange. Why Do Languages Such as C and C++ Not Have Garbage Collection While Java Does? 2016. <http://programmers.stackexchange.com/questions/113177/why-do-languages-such-as-c-and-c-not-have-garbage-collection-while-java-does> (accessed January 11, 2016).

Puder, A., and Romer, K. *MICO: An Open Source CORBA Implementation*, Morgan-Kaufmann, San Francisco, CA, 2000.

Puder, A., Romer, K., and Pilhofer, F. *Distributed Systems Architecture: A Middleware Approach*, Morgan-Kaufman, San Francisco, CA, 2006.

Sami, A. WCF Comparison with Web Services and .NET Remoting. CodeProject. 2010. <http://www.codeproject.com/Articles/45698/WCF-Comparison-with-Web-Services-and-NET-Remoting> (accessed March 18, 2016).

Schmidt, D.C. Dynamic CORBA: Part 1: The Dynamic Invocation Interface. Dr. Dobb's. 2002. <http://www.drddobbs.com/dynamic-corba-part-1-the-dynamic-invocat/184403833?pgno=2> (accessed January 31, 2016).

Schmidt, D.C. Overview of ACE. 2011. <http://www.cs.wustl.edu/~schmidt/ACE-overview.html> (accessed January 14, 2016).

Schmidt, D. TAO Interoperable Naming Service. http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/INS.html#oir (accessed January 19, 2016).

- Schmidt, D.C., and Vinoski, S. The History of OMG C++ Mapping, C/C++ Users Journal C++ Experts Forum. 2000. <https://www.dre.vanderbilt.edu/~schmidt/PDF/cuj-1.pdf> (accessed January 12, 2016).
- Schuster, W. Andrew Watson on the State of OMG, UML, CORBA, DDS. 2012. <http://www.infoq.com/interviews/watson-omg> (accessed November 28, 2015).
- Siegel, J. *CORBA 3: Fundamentals and Programming*, Wiley, Hoboken, NJ, 2000.
- Stack Overflow. Assigning a String of Characters to a Char Array. 2012. <http://stackoverflow.com/questions/10088661/assigning-a-string-of-characters-to-a-char-array> (accessed January 14, 2016).
- Stack Overflow. Difference between Application Manifest and Assembly Manifest. 2010. <http://stackoverflow.com/questions/3476089/difference-between-application-manifest-and-assembly-manifest> (accessed March 24, 2016).
- Stack Overflow. Difference between Java Bean and EJBs. 2010. <http://stackoverflow.com/questions/2460048/difference-between-java-bean-and-enterprise-java-beans> (accessed January 31, 2016).
- Stack Overflow. Get the Client Orb Address and Port with Use of IIOP. 2014. <http://stackoverflow.com/questions/14479816/corba-get-the-client-orb-address-and-port-with-use-of-iiop> (accessed January 19, 2016).
- Stack Overflow. IAsyncResult.AsyncWaitHandle.WaitOne() Completes Ahead of Callback. 2010. <http://stackoverflow.com/questions/4099318/iasyncresult-asyncwaithandle-waitone-completes-ahead-of-callback> (accessed March 26, 2016).
- Stack Overflow. .Net Remoting, Issue with Firewall. 2010. <http://stackoverflow.com/questions/2297663/net-remoting-issue-with-firewall> (accessed March 26, 2016).
- Stack Overflow. Servlet versus RESTful. 2011. <http://stackoverflow.com/questions/7874695/servlet-vs-restful> (accessed February 10, 2016).
- Stack Overflow. What Is the Difference between Managed C++ and C++/CLI. 2010. <http://stackoverflow.com/questions/2443811/what-is-the-difference-between-managed-c-and-c-cli> (accessed March 18, 2016).
- Tutorials Point. Callbacks—EJB. 2016. http://www.tutorialspoint.com/ejb/ejb_callbacks.htm (accessed February 8, 2016).
- Tutorials Point. EJB Tutorial. 2006. <http://www.tutorialspoint.com/ejb/> (accessed February 7, 2016).
- ul Hassan, S.N. .NET Remoting with an Easy Example. CodeProject. 2006. <http://www.codeproject.com/Articles/14791/NET-Remoting-with-an-easy-example> (accessed March 26, 2016).
- Vogel, A., and Duddy, K. *Java Programming with CORBA*, 2nd Edition, 1998.
- Whiley, Java versus C++ Performance. 2012. <http://whiley.org/2012/08/13/java-versus-c-performance/> (accessed January 11, 2016).
- Wikipedia. Create, Read, Update, and Delete. 2016. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete (accessed February 10, 2016).

5. Methodology

Chapter Two

Geometric models and optimization

This chapter presents various geometric objects and geometric optimization problems that have strong connections with motion coordination. Basic geometric notions such as polytopes, centers, partitions, and distances are ubiquitous in cooperative strategies, coordination tasks, and the interaction of robotic networks with the physical environment. The notion of Voronoi partition finds application in diverse areas such as wireless communications, signal compression, facility location, and mesh optimization. Proximity graphs provide a natural way to mathematically model the network interconnection topology resulting from the agents' sensing and/or communication capabilities. Finally, multicenter functions play the role of aggregate objective functions in geometric optimization problems. We introduce these concepts here in preparation for the later chapters.

The chapter is organized as follows. We begin by presenting basic geometric constructions. This gives way to introduce the notion of proximity graphs along with numerous examples. The next section of the chapter presents geometric optimization problems and multicenter functions, paying special attention to the characterization of their smoothness properties and critical points. We end the chapter with three sections on, respectively, bibliographic notes, proofs of the results presented in the chapter, and exercises.

2.1 BASIC GEOMETRIC NOTIONS

In this section, we gather some classical geometric

constructions that will be invoked regularly throughout the book.

2.1.1 Polygons and polytopes

For $p, q \in \mathbb{R}^d$, we let $]p, q[= \{\lambda p + (1 - \lambda)q \mid \lambda \in]0, 1[\}$ and $[p, q] = \{\lambda p + (1 - \lambda)q \mid \lambda \in [0, 1] \}$ denote the *open segment* and *closed segment*, with extreme

points p and q , respectively. We let $H_{p,q} = \{x \in \mathbb{R}^d \mid \|x - p\| \leq \|x - q\|\}$ denote the *closed halfspace* of $\mathbb{R}^d$ of points closer (in Euclidean distance) to p than to q . In the plane, we often refer to a halfspace as a *halfplane*.

As seen in Section 1.2, a set $S \subset \mathbb{R}^d$ is *convex* if, for any two points p, q

in S , the closed segment $[p, q]$ is contained in S . The *convex hull* of a set

is the smallest (with respect to the inclusion) convex set that contains it. We denote the convex hull of S by $\text{co}(S)$.

For $S = \{p_1, \dots, p_n\}$ finite, the convex hull can be explicitly described as follows:

$$\text{co}(S) = \left\{ \sum_{i=1}^n \lambda_i p_i \mid \lambda_i \geq 0 \text{ and } \sum_{i=1}^n \lambda_i = 1 \right\}.$$

Given p and q in $\mathbb{R}^d$ and a convex closed set $Q \subset \mathbb{R}^d$ with $p \in Q$ (see Figure 2.1), define the *from-to-inside* function by

$$\text{fti}(p, q, Q) = \begin{cases} q, & \text{if } q \in Q, \\ [p, q] \cap \partial Q, & \text{if } q \notin Q. \end{cases}$$

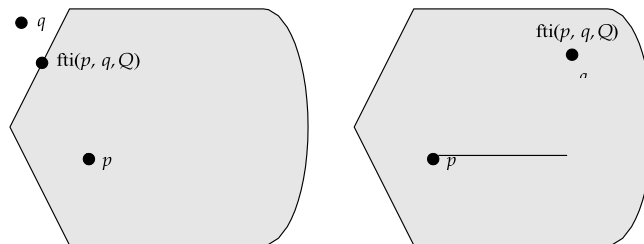


Figure 2.1 An illustration of the from-to-inside function fti .

The function fti selects the point in the closed segment $[p, q]$ which is at the same time closest to q and inside Q . Note that $\text{fti}(p, q, Q)$ depends continuously on p and q .

A *polygon* is a set in $\mathbb{R}^2$ whose boundary is the union of a finite number of closed segments. A polygon is *simple* if its boundary, regarded as a curve,

is not self-intersecting. We will only consider simple polygons. The closed segments composing the boundary of a polygon are called *edges*, and points resulting from the pairwise intersection between consecutive edges are called *vertices*. A convex polygon can be written as:

- (i) the convex hull of its set of vertices; or
- (ii) the intersection of halfplanes defined by its edges.

Two vertices whose open segment is contained in the interior of the polygon define a *diagonal*. To each vertex of a polygon we associate an *interior* and an *exterior angle*. A vertex is *strictly convex* (resp. *strictly nonconvex*) if its interior angle is strictly smaller (resp. greater) than π radians. A polygon is nonconvex if it has at least one strictly concave vertex. The *perimeter* of a polygon is the length of its boundary, that is, the sum of the lengths of its edges. A polytope is the generalization of the notion of polygon to

$\mathbb{R}^d$, for $d \geq 3$. In this book, we will not consider nonconvex polytopes in dimension larger than 2. As for convex polygons, a (*convex*) *polytope* in $\mathbb{R}^d$ can be defined as either the convex hull of a finite set of points in $\mathbb{R}^d$ or the bounded intersection of a finite set of halfspaces. A $d - 1$ *face* (or a *facet*)

of a polytope is the intersection between the polytope and the boundary of a closed halfspace that defines the polytope. A $d - 2$ face is a $d - 2$ face of a facet of the polytope. The faces of dimensions 0, 1, and $d - 1$ are called *vertices*, *edges*, and *faces*, respectively. For a convex polytope Q , we will refer to them as $\text{Ve}(Q)$, $\text{Ed}(Q)$, and $\text{Fa}(Q)$, respectively.

2.1.2 Nonconvex geometry

In this section, we gather some basic notions on nonconvex geometry. We consider environments that include nonconvex polygons as a particular case.

We begin with some visibility notions. Given $S \subset \mathbb{R}^d$, two points p, q are *visible* to each other if the closed segment $[p, q]$ is contained in S . The

visibility set $\text{Vi}(p; S)$ is the set of all points in S visible from p . Given $r > 0$, the *range-limited visibility set* $\text{Vi}_{\text{disk}}(p; S) = \text{Vi}(p; S) \cap \overline{B(p, r)}$ is the set of all points in S within a distance r and visible from p . The set S is *star-shaped* if there exists $p \in S$ such that $\text{Vi}(p; S) = S$. The *kernel set* of S is comprised of

all the points with this property, that is, $\text{kernel}(S) = \{p \in S \mid \forall i(p; S) = S\}$. Trivially, any convex set is star-shaped. Given $\delta \in \mathbb{R}_{>0}$, the δ -contraction of S is the set $S_\delta = \{p \in S \mid \text{dist}(p, \partial S) \geq \delta\}$. Note that if two points $p, q \in S$ are visible to each other in S_δ , then any point within distance δ of p and any point within distance δ of q are visible to each other. Figure 2.2 illustrates these visibility notions.

Next, we introduce various concavity notions. Given $S \subset \mathbb{R}^d$ connected and closed, $p \in \partial S$ is *strictly concave* if, for any $\varepsilon \in \mathbb{R}_{>0}$, there exist $q_1, q_2 \in B(p, \varepsilon) \cap \partial S$ such that $[q_1, q_2] \not\subset S$. This definition coincides with

the notion of strictly concave vertex when the set S is a polygon. A *strict concavity* of S is either an isolated strictly concave point or a concave arc, that is, a connected set of strictly concave points. An *allowable environment* $S \subset \mathbb{R}^2$ is a set that satisfies the following properties: it is closed, simply connected, has a finite number of strict concavities, and its boundary can

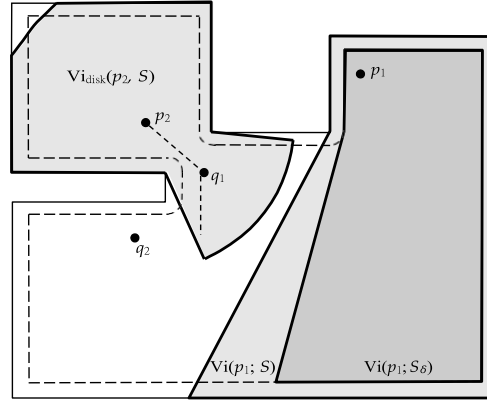


Figure 2.2 An illustration of various visibility notions. The visibility set $Vi(p_1; S)$ from p_1 in S , the visibility set $Vi(p_1; S_\delta)$ from p_1 in S_δ , and the range-limited visibility set $Vi_{\text{disk}}(p_2; S)$ from p_2 in S are depicted in light gray. The dashed curve in the interior of S corresponds to the boundary of the δ -contraction of S . The points p_2 and q_1 are visible to each other in S_δ . The points q_1 and q_2 are visible to each other in S , but they are not visible to each other in S_δ .

be described by a continuous and piecewise continuously differentiable curve which is not differentiable at most at a finite number of points. Figure 2.3 shows a sample allowable environment. Given an allowable environment S , let a point v belonging to a concave arc have the property that the boundary

of S is continuously differentiable at v . The *internal tangent halfplane* $H_S(v)$ is the closed halfplane whose boundary is tangent to ∂S at v and whose

interior does not contain any points of the strict concavity (see Figure 2.3).

The following result presents an interesting property of allowable environments. Its proof is left to the reader.

Lemma 2.1 (Contraction of allowable environments). *Given an allowable environment S , the δ -contraction S_δ is also allowable for sufficiently small $\delta \in \mathbb{R}_{>0}$ and does not have isolated strictly concave points. Furthermore, the boundary of S_δ is continuously differentiable at the concavities.*

Lemma 2.1 implies that the internal tangent halfplane is well-defined at any strict concavity of the δ -contraction

S_δ .

A set $S \subseteq X$ is *relatively convex* in $X \subseteq \mathbb{R}^d$ if, for any two points p, q in S , the shortest curve in X that connects p and q is contained in S . Relatively convex sets in $\mathbb{R}^d$ are just convex sets. The *relative convex hull* of a set S in X is the smallest (with respect to the operation of inclusion)

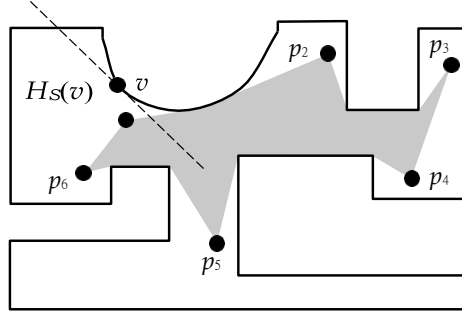


Figure 2.3 An allowable environment S . The curved portion of the boundary is a concave arc. The vertices whose interior angle is $3\pi/2$ radians are isolated strictly concave points. The relative convex hull of $\{p_1, \dots, p_6\}$ in S is depicted in light gray. Finally, the dashed line represents the boundary of the internal tangent halfplane $H_S(v)$ tangent to ∂S at v .

relatively convex set in X that contains S (see Figure 2.3). We denote the relative convex hull of S in X by $\text{rco}(S; X)$. The (*relative*) *perimeter* of S in X is the length of the shortest measurable closed curve contained in X that encloses S .

2.1.3 Geometric centers

Let $X = \mathbb{R}^d$, $X = \mathbb{S}^d$ or $X = \mathbb{R}^{d_1} \times \mathbb{S}^{d_2}$, $d = d_1 + d_2$. Recall our convention (cf., Section 1.1.2) that, unless otherwise noted, $\mathbb{R}^d$ is endowed with the Euclidean distance, $\mathbb{S}^d$ is endowed with the geodesic distance, and $\mathbb{R}^{d_1} \times \mathbb{S}^{d_2}$ is endowed with the Cartesian product distance (dist_2 , dist_g).

The *circumcenter* of a bounded set $S \subset X$, denoted by $\text{CC}(S)$, is the center of the closed ball of minimum radius that contains S . The *circumradius* of S , denoted by $\text{CR}(S)$, is the radius of this ball¹. The circumcenter is always unique.

The computation of the circumcenter and the circumradius of a polytope $Q \subset \mathbb{R}^d$ is a strictly convex problem and, in particular, a quadratically constrained

linear program in p (the center) and r (the radius). It consists of minimizing the radius r of the ball centered at p subject to the constraints that the distance between q and each of the polygon vertices is smaller than

¹Note that the definition of circumcenter given here is in general different from the classical notion of circumcenter of a triangle, that is, the center of the circle passing through the three vertices of the triangle.

or equal to r . Formally, the problem can

be expressed as minimize r ,

$$\text{subject to } \|q - p\|^2 \leq r^2, \text{ for all } q \in \text{Ve}(Q). \quad (2.1.1)$$

Next, we summarize some useful properties of the circumcenter in Euclidean space; see Exercise E2.1 for their proofs. In the following result, for $S \subset \mathbb{R}^d$ with $d = 1$, we let $\text{Ve}(\text{co}(S))$ denote the set of extreme points of the interval $\text{co}(S)$.

Lemma 2.2 (Properties of the circumcenter in Euclidean space).

Let $S = \{p_1, \dots, p_n\} \in \mathbb{R}^d$ with $n \geq 2$. The following properties hold:

- (i) $\text{CC}(S) \in \text{co}(S) \setminus \text{Ve}(\text{co}(S))$; and
- (ii) if $p \in \text{co}(S) \setminus \text{CC}(S)$ and $r \in \mathbb{R}_{>0}$ are such that $S \subset \overline{B}(p, r)$, then $\overline{B}(p, \text{CC}(S))$ has a nonempty intersection with $\overline{B}(p, r)$ for all $q \in \text{co}(S)$.

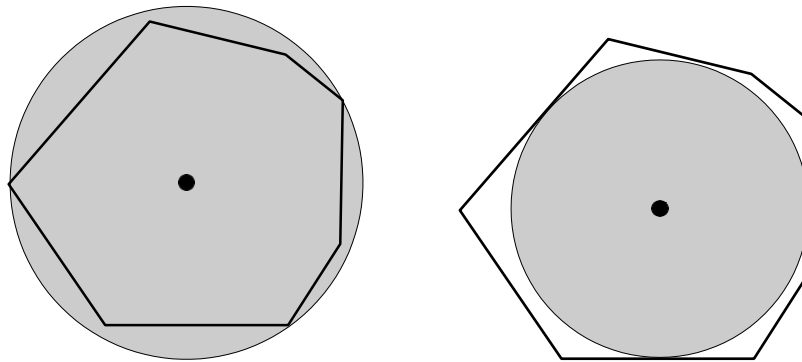


Figure 2.4 The circumcenter and circumradius (left), and incenter and inradius (right) of a convex polygon.

Given $X = \mathbb{R}^d$, $X = S^d$ or $X = \mathbb{R}^{d_1} \times S^{d_2}$, $d = d_1 + d_2$, the *incenter*, or *Chebyshev center* of a compact set $S \subset X$, denoted by $\text{IC}(S)$, is the set

containing the centers of all closed balls of maximum radius contained in S . The *inradius* of S , denoted by $\text{IR}(S)$, is the common radius of any of these balls.

The computation of the incenter and the inradius of a polytope Q in $\mathbb{R}^d$ is a convex problem and, in particular, a linear program in p and r . It consists of maximizing the radius r of the ball centered at p subject to the constraints that the distance between p and each of the polytope facets is greater than or equal to r . Formally, the problem can be expressed as follows. For each

$f \in \text{Fa}(Q)$, select a point $q_f \in Q$ belonging to f .

Then, we set maximize r ,

subject to $(q_f - p) \cdot n_{\text{out}} \geq r$, for all $f \in \text{Fa}(Q)$,

(2.1.

2) where n_{out} denotes the normal to the face f pointing toward the exterior of the polytope. The incenter of a polytope is not necessarily unique (consider, for instance, the case of a rectangle).

In Euclidean space, $X = \mathbb{R}^d$, we refer to a bounded measurable function $\phi : \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$ as a *density* on $\mathbb{R}^d$. The (*generalized*) *area* and the *centroid* (also called center of mass) of a bounded measurable set $S \subset \mathbb{R}^d$ with respect to ϕ , denoted by $A_\phi(S)$ and $\text{CM}_\phi(S)$ respectively, are given by

$$A_\phi(S) = \int_S \phi(q) dq, \quad \text{CM}_\phi(S) = \frac{1}{A_\phi(S)} \int_S q \phi(q) dq.$$

When the function ϕ that is being used is clear from the context, we simply refer to the area and the centroid of S . The centroid can alternatively be defined as follows. Define the *polar moment of inertia* of S about $p \in S$ by

$$J_\phi(S, p) = \int_S \|q - p\|^2 \phi(q) dq.$$

Then, the centroid of S is precisely the point $p \in S$ that minimizes the polar moment of inertia of S about p . This can be easily seen from the Parallel Axis Theorem (Hibbeler, 2006), which states that

$$J_\phi(S, p) = J_\phi(S, \text{CM}_\phi(S)) + A_\phi(S) \|p - \text{CM}_\phi(S)\|^2.$$

2

Remark 2.3 (Computation of geometric centers in the plane). The circumcenter, incenter, and centroid of a polygon can be computed in several ways. A simple procedure to compute the circumcenter consists of

enumerating all pairs and triplets of vertices of the polygon, computing the centers and radii of the balls passing through them, and selecting the ball with the smallest radius that encloses the polygon. An alternative, more efficient, way of computing the circumcenter is to use the formulation (2.1.1). A convex quadratically constrained linear program is a particular case of a semidefinite-quadratic-linear program (SQLP). Several freely available numerical packages exist to solve SQLP problems; for example, SDPT3 (Tutuncu et al., 2003). The computation of the incenter set of a polygon can be performed via linear programming using the formulation (2.1.2). Finally, the centroid of a polygon can be computed with any numerical routine that accurately approximates the integral of a function over a planar domain. •

2.1.4 Voronoi and range-limited Voronoi partitions

A *partition* of a set S is a subdivision of S into connected subsets that are disjoint except for their boundary. Formally, a partition of S is a collection of closed connected sets $\{W_1, \dots, W_m\} \subset P(S)$ that verify

$$S = \bigcup_{i=1}^m W_i \quad \text{and} \quad \text{int}(W_j) \cap \text{int}(W_k) = \emptyset,$$

for $j, k \in \{1, \dots, m\}$.

Definition 2.4 (Voronoi partition). Given a distance function $\text{dist} : X \times X \rightarrow \mathbb{R}_{\geq 0}$, a set $S \subset X$ and n distinct points $\mathbf{P} = \{p_1, \dots, p_n\}$ in S , the *Voronoi partition* of S generated by $\mathbf{P}$ is the collection of sets $V(\mathbf{P}) =$

$\{V_1(\mathbf{P}), \dots, V_n(\mathbf{P})\} \subset P(S)$ defined by, for each $i \in \{1, \dots, n\}$,

$$V_i(\mathbf{P}) = \{q \in S \mid \text{dist}(q, p_i) \leq \text{dist}(q, p_j), \text{ for all } p_j \in \mathbf{P} \setminus \{p_i\}\}. \quad \bullet$$

In other words, $V_i(\mathbf{P})$ is the set of the points of S that are closer to p_i than to any of the other points in $\mathbf{P}$. We refer to $V_i(\mathbf{P})$ as the *Voronoi cell* of p_i . Unless explicitly noted otherwise, we compute the Voronoi partition according to the following conventions:

- for $X = \mathbb{R}^d$, with respect to the Euclidean distance;
- for $X = S^d$, with respect to the geodesic distance; and
- for $X = \mathbb{R}_{X_1}^{d_1} \times \mathbb{R}_{X_2}^{d_2}$, $d_1 + d_2 = d$, with respect to the Cartesian product distance determined by dist_2 on $\mathbb{R}_{X_1}^{d_1}$ and dist_g on S^{d_2} .

Figure 2.5 shows an example of the Voronoi partition of the circle generated by five points. In the Euclidean case, the Voronoi cell of p_i is equal to the intersection of half-spaces determined by p_i and the other locations in $\mathbf{P}$, and as such it is a convex polytope. The left plot in Figure 2.6 shows an example of the Voronoi partition of a convex polygon

generated by 40 points.

Definition 2.5 (*r*-limited Voronoi partition). Given a distance function $\text{dist} : X \times X \rightarrow \mathbb{R}_{\geq 0}$, a set $S \subset X$, n distinct points $\mathbf{P} = \{p_1, \dots, p_n\}$ in S , and a positive real number $r \in \mathbb{R}_{>0}$, the *r*-limited Voronoi partition inside S generated by $\mathbf{P}$ is the collection of sets $\{V_1(\mathbf{P}), \dots, V_n(\mathbf{P})\}$ defined by

$$V_{i,r}(\mathbf{P}) = V_i(\mathbf{P}) \cap \overline{B(p_i, r)}, \quad i \in \{1, \dots, n\}. \quad \cdot$$

Note that the *r*-limited Voronoi partition inside S is precisely the Voronoi partition of the set $\bigcup_{i=1}^n \overline{B(p_i, r)} \cap S$. We will refer to $V_{i,r}(\mathbf{P})$ as the *r*-limited

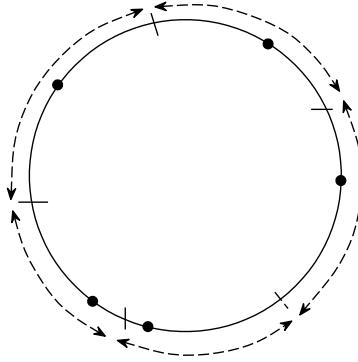


Figure 2.5 Voronoi partition of the circle generated by five points. The dashed segments correspond to the Voronoi cells of each individual point.

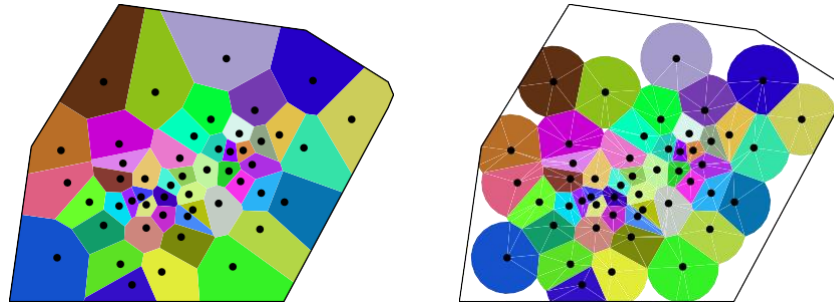


Figure 2.6 Voronoi partition of a convex polygon (left) and r -limited Voronoi partition inside a convex polygon (right) generated by 40 points.

Voronoi cell of p_i . The right-hand plot in Figure 2.6 shows an example of the r -limited Voronoi partition inside a convex polygon generated by 40 points.

Let $X = \mathbb{R}^d$, $X = S^d$ or $X = \mathbb{R}^{d_1} \times S^{d_2}$, $d = d_1 + d_2$. Given a density φ

on X , a set of n distinct points $P = \{p_1, \dots, p_n\}$ in $S \subset X$ is:

- (i) A *centroidal Voronoi configuration* if each point is the centroid of its own Voronoi cell, that is, $p_i = \text{CM}_\phi(V_i(P))$.
- (ii) An *r -limited centroidal Voronoi configuration*, for $r \in \mathbb{R}_{>0}$, if each point is the centroid of its own r -limited Voronoi cell, that is, $p_i = \text{CM}_\phi(V_{i,r}(P))$. If $r \geq \text{diam}(S)$, then an r -limited centroidal Voronoi configuration is a centroidal Voronoi configuration.
- (iii) A *circumcenter Voronoi configuration* if each point is the circumcenter of its own Voronoi cell, that is, $p_i = \text{CC}(V_i(P))$.
- (iv) An *incenter Voronoi configuration* if each point is an incenter of its

own Voronoi cell, that is, $p_i \in \text{IC}(V_i(\mathbf{P}))$.

Figure 2.7 illustrates of the various notions of center Voronoi configurations.

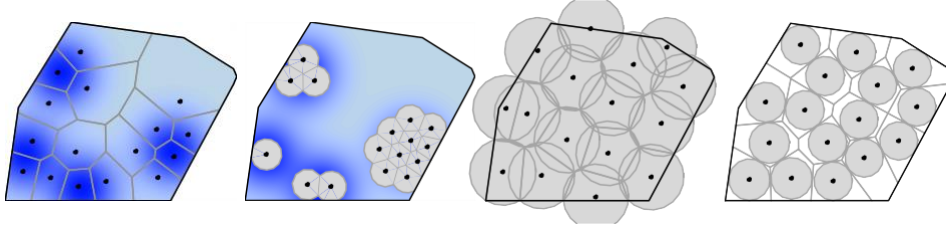


Figure 2.7 From left to right, centroidal, r -limited centroidal, circumcenter, and incenter Voronoi configurations composed by 16 points in a convex polygon. Darker blue-colored areas correspond to higher values of the density φ .

2.2 PROXIMITY GRAPHS

Roughly speaking, a proximity graph is a graph whose vertex set is a set of distinct points and whose edge set is a function of the relative locations of the point set. Proximity graphs appear in computational geometry. In this section, we study this important notion in detail following the presentation by [Cortés et al. \(2005\)](#).

Definition 2.6 (Proximity graph). Assume that X is a d -dimensional space chosen among $\mathbb{R}^d$, $\mathbb{S}^d$, and the Cartesian products $\mathbb{R}^{d_1} \times \mathbb{S}^{d_2}$, for some $d_1 + d_2 = d$. For a set $S \subset X$, let $\mathcal{G}(S)$ be the set of all undirected graphs whose vertex set is an element of $\mathcal{F}(S)$. A *proximity graph* $G : \mathcal{F}(S) \rightarrow \mathcal{G}(S)$ associates to a set of distinct points $\mathbf{P} = \{p_1, \dots, p_n\} \subset S$ an undirected graph with vertex set $\mathbf{P}$ and whose edge set is given by $E_G(\mathbf{P}) \subseteq \{(p, q) \in \mathbf{P} \times \mathbf{P} \mid p \neq q\}$.

.

Note that in a proximity graph a point cannot be its own neighbor. From this definition, we observe that the distinguishing feature of proximity graphs is that their

edge sets change with the location of their vertices. It is also possible to define proximity graphs that associate to each point set a digraph, but we will not consider them here.

Examples of proximity graphs on X , where we recall that $\text{dist} = \text{dist}_2$ if $X = \mathbf{R}^d$, $\text{dist} = \text{dist}_g$ if $X = S^d$, and $\text{dist} = (\text{dist}_2, \text{dist}_g)$ if $X = \mathbf{R}^{d_1} \times S^{d_2}$, include the following:

- (i) The *complete graph* G_{cplt} where any two points are neighbors. When convenient, we may view the complete graph as weighted by assigning the weight $\text{dist}(p_i, p_j)$ to the edge $(p_i, p_j) \in E_{\text{cplt}}(\mathbf{P})$.
- (ii) The *r-disk graph* $G_{\text{disk}}(r)$, for $r \in \mathbf{R}_{>0}$, where two points are neigh-

bors if they are located within a distance r , that is, $(p_i, p_j) \in E_{G_{\text{disk}}(r)}(\mathbf{P})$ if $\text{dist}(p_i, p_j) \leq r$.

- (iii) The *Delaunay graph* G_D , where two points are neighbors if their corresponding Voronoi cells intersect, that is, $(p_i, p_j) \in E_{G_D}(\mathbf{P})$ if $V_i(\mathbf{P}) \cap V_j(\mathbf{P}) \neq \emptyset$.
- (iv) The *r -limited Delaunay graph* $G_{LD}(r)$, for $r \in \mathbb{R}_{>0}$, where two points are neighbors if their corresponding r -limited Voronoi cells intersect, that is, $(p_i, p_j) \in E_{G_{LD}(r)}(\mathbf{P})$ if $V_{i,r}(\mathbf{P}) \cap V_{j,r}(\mathbf{P}) \neq \emptyset$.
- (v) The *relative neighborhood graph* G_{RN} , where two points are neighbors if their associated open lune (cf. Section 1.1.2) does not contain any point in $\mathbf{P}$, that is, $(p_i, p_j) \in E_{G_{RN}}(\mathbf{P})$ if, for all $p_k \in \mathbf{P}$, $k \notin \{i, j\}$

$$p_k \notin B(p_i, \text{dist}(p_i, p_j)) \cap B(p_j, \text{dist}(p_i, p_j)).$$

Figure 2.8 shows examples of these proximity graphs in the plane.

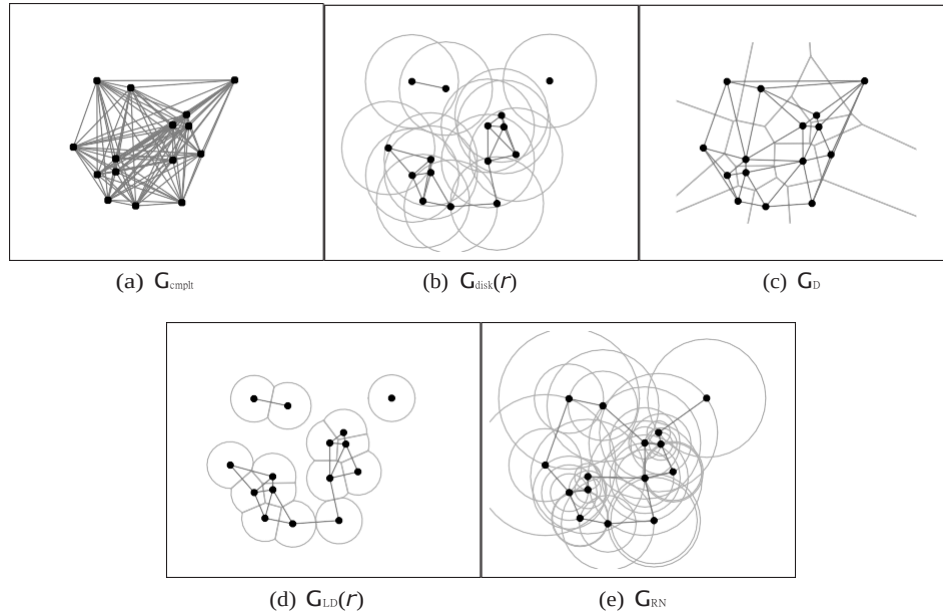


Figure 2.8 Proximity graphs in $\mathbb{R}^2$. From left to right, in the first row, complete, r -disk, and Delaunay, and in the second row, r -limited Delaunay and relative neighborhood for a set of 15 points. When appropriate, the geometric objects determining the edge relationship are plotted in lighter gray.

Additional examples of proximity graphs in the Euclidean space include the following:

104

673

- (vi) The *Gabriel graph* G_G , where two points are neighbors if the ball centered at their midpoint and passing through both of them does not contain any point in P , that is, $(p_i, p_j) \in E_{G_G}(P)$ if $\exists p_k \in P$ such that $\|p_k - \frac{p_i + p_j}{2}\| < \frac{\|p_i - p_j\|}{2}$ for all $p_k \in P$.
- (vii) The *r-disk graph* $G_{\text{disk}}(r)$, for $r \in \mathbb{R}_{>0}$, where two points are neighbors if they are located within L^∞ -distance r , that is, $(p_i, p_j) \in E_{G_{\text{disk}}(r)}(P)$ if $\text{dist}_\infty(p_i, p_j) \leq r$.
- (viii) The *Euclidean minimum spanning tree* of a proximity graph G , denoted by $G_{\text{EMST}, G}$, that assigns to each P a minimum-weight spanning tree (cf., Section 1.4.4.4) of $G(P)$ with weighted adjacency matrix $a_{ij} = \|p_i - p_j\|^2$, for $(p_i, p_j) \in E_G(P)$. If $G(P)$ is not connected, then $G_{\text{EMST}, G}(P)$ is the union of Euclidean minimum spanning trees of its connected components. When G is the complete graph, we simply denote the Euclidean minimum spanning tree by G_{EMST} .
- (ix) the *visibility graph* $G_{\text{vis}, Q}$ in an allowable environment Q in $\mathbb{R}^2$, where two points are neighbors if they are visible to each other, that is, $(p_i, p_j) \in E_{G_{\text{vis}, Q}}(P)$ if the closed segment $[p_i, p_j]$ from p_i to p_j is contained in Q .
- (x) The *range-limited visibility graph* $G_{\text{vis-disk}, Q}$ in an allowable environment Q in $\mathbb{R}^2$, where two points are neighbors if they are visible to each other and their distance is no more than r , that is, $(p_i, p_j) \in E_{G_{\text{vis-disk}, Q}}(P)$ if $(p_i, p_j) \in E_{G_{\text{vis}, Q}}(P)$ and $(p_i, p_j) \in E_{G_{\text{disk}}(r)}(P)$.

Figure 2.9 shows examples of these proximity graphs in the plane; Figure 2.10 shows examples of these proximity graphs in a planar nonconvex environment; and Figure 2.11 shows example graphs in three-dimensions.

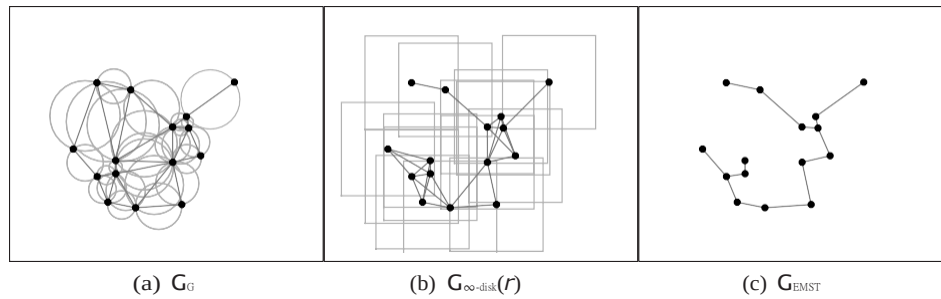


Figure 2.9 Proximity graphs in $\mathbb{R}^2$. From left to right, Gabriel graph, r - ∞ -disk graph, and Euclidean minimum spanning tree for 15 points. In two images, the geometric objects determining the edge relationship are plotted in light gray.

As for standard graphs, let us alternatively describe the edge set by means of the sets of neighbors of the individual graph vertices. To each proximity

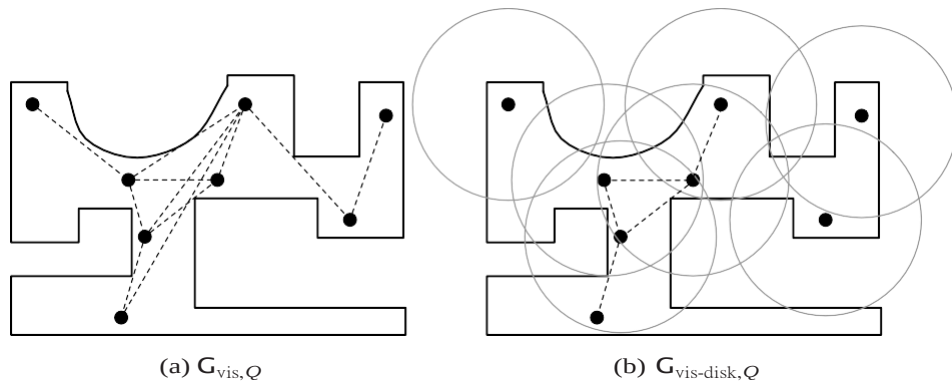


Figure 2.10 The visibility and range-limited visibility graphs for 8 agents in an allowable environment. The geometric objects determining the edge relationship are plotted in light gray.

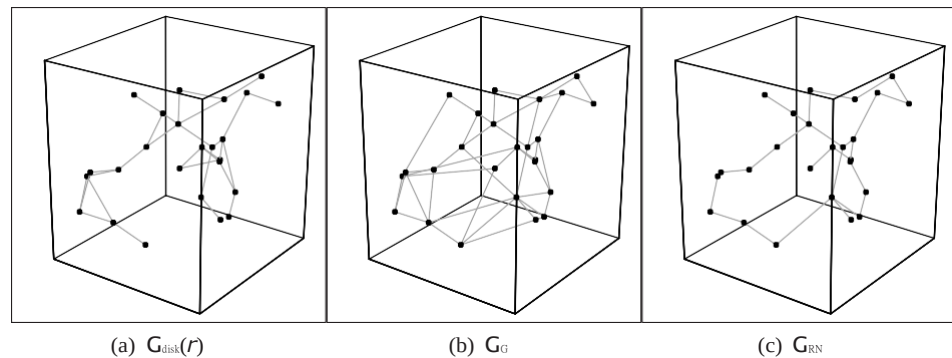


Figure 2.11 Proximity graphs in $\mathbb{R}^3$. From left to right, r -disk, relative neighborhood, and Gabriel graphs for a set of 25 points.

graph G , each $p \in X$ and each $P = \{p_1, \dots, p_n\} \in F(X)$, we associate the set of neighbors map $N_G : X \times F(X) \rightarrow F(X)$ defined by

$$N_G(p, P) = \{q \in P \mid (p, q) \in E_G(P \cup \{p\})\}.$$

Typically, p is a point in P , but the definition is well-posed for any $p \in X$. Under the assumption that P does not contain repeated elements, the definition will not lead to counterintuitive interpretations later. Given $p \in X$, it is convenient to define the map $N_{G,p} : F(X) \rightarrow F(X)$ by $N_{G,p}(P) = N_G(p, P)$.

A proximity graph G_1 is a *subgraph* of a proximity

$$G \subseteq G \quad P \quad G \quad P \quad 676 \quad P$$

graph G_2 , denoted $G_1 \subset_2 G_2$, if $G_1(X)$ is a subgraph of $G_2(X)$ for all $X \in F(X)$. The following result, whose proof is given in Section 2.5.1, summarizes the subgraph relationships in the Euclidean case among the various proximity graphs introduced above.

Theorem 2.7 (Subgraph relationships among some standard proximity graphs on $\mathbf{R}^d$). *For $r \in \mathbf{R}_{>0}$, the following statements hold:*

- (i) $G_{\text{EMST}} \subset G_{\text{RN}} \subset G_G \subset G_D$; and
- (ii) $G_G \cap G_{\text{disk}}(r) \subset G_{\text{LD}}(r) \subset G_D \cap G_{\text{disk}}(r)$.

Note that the inclusion $G_{\text{LD}}(r) \subset G_D \cap G_{\text{disk}}(r)$ is in general strict; this counterintuitive fact is discussed in Exercise E2.3. Additionally, since G_{EMST} is by definition connected, Theorem 2.7(i) implies that G_{RN} , G_G , and G_D are connected. The connectivity properties of $G_{\text{disk}}(r)$ are characterized in the following result.

Theorem 2.8 (Connectivity properties of some standard proximity graphs on $\mathbf{R}^d$). *For $r \in \mathbf{R}_{>0}$, the following statements hold:*

- (i) $G_{\text{EMST}} \subset G_{\text{disk}}(r)$ if and only if $G_{\text{disk}}(r)$ is connected; and
- (ii) $G_{\text{EMST}} \cap G_{\text{disk}}(r)$, $G_{\text{RN}} \cap G_{\text{disk}}(r)$, $G_G \cap G_{\text{disk}}(r)$ and $G_{\text{LD}}(r)$ have the same connected components as $G_{\text{disk}}(r)$ (i.e., for all point sets $P \subset \mathbf{R}^d$, all graphs have the same number of connected components consisting of the same vertices).

The proof of this theorem is given in Section 2.5.1. Note that in Theorem 2.8, fact (ii) implies (i).

2.2.1 Spatially distributed proximity graphs

We now consider the following loosely stated question: When does a given proximity graph encode sufficient information to compute another proximity graph? For instance, if a node knows the position of its neighbors in the complete graph (i.e., of every other node in the graph), then it is clear that the node can compute its neighbors with respect to any proximity graph.

Let us formalize this idea. A proximity graph G_1 is *spatially distributed over*

a proximity graph G_2 if, for all $p \in P$,

$$N_{G_1,p}(P) = N_{G_1,p} \cap N_{G_2,p}(P) ,$$

that is, any node informed about the location of its neighbors with respect to G_2 can compute its set of neighbors with respect to G_1 .

Clearly, any proximity graph is spatially distributed over the complete graph. It is straightforward to deduce that if G_1 is spatially distributed over G_2 , then G_1 is a subgraph of G_2 . The converse is in general not true. For instance, $G_D \cap G_{\text{disk}}(r)$ is a subgraph of $G_{\text{disk}}(r)$, but $G_D \cap G_{\text{disk}}(r)$ is not spatially distributed over $G_{\text{disk}}(r)$; see Exercise [E2.4](#).

The following result identifies proximity graphs which are spatially distributed over $G_{\text{disk}}(r)$.

Proposition 2.9 (Spatially distributed graphs over the disk graph). *The proximity graphs $G_{\text{RN}} \cap G_{\text{disk}}(r)$, $G_{\text{G}} \cap G_{\text{disk}}(r)$, and $G_{\text{LD}}(r)$ are spatially distributed over $G_{\text{disk}}(r)$.*

Remark 2.10 (Computation of the Delaunay graph over the r -disk graph). In general, for a fixed $r \in \mathbb{R}_{>0}$, G_{D} is not spatially distributed over $G_{\text{disk}}(r)$. However, for a given $P \in \mathcal{F}(X)$, it is always possible to find r such that $G_{\text{D}}(P)$ is spatially distributed over $G_{\text{disk}}(r)(P)$. This is a consequence of the following observations. Given $P \in \mathcal{F}(X)$, define the convex sets

$$W(p_i, r) = B(p_i, r) \cap \bigcap_{\|p_i - p_j\| \leq r} H_{p_i, p_j}, \quad i \in \{1, \dots, n\},$$

where we recall that $H_{p, x}$ is the half-space of points q in $\mathbb{R}^d$ with the property that $\|q - p\|_2 \leq \|q - x\|_2$. Note that the intersection $B(p_i, r) \cap V_i$ is a subset of $W(p_i, r)$. Provided that r is twice as large as the maximum distance between p_i and the vertices of $W(p_i, r)$, then all Delaunay neighbors of p_i are within distance r from p_i . Equivalently, the half-space $H_{p_i, p}$ determined by p_i and a point p outside $B(p_i, r)$ does not intersect $W(p_i, r)$. Therefore, the equality $V_i = W(p_i, r)$ holds. For node $i = 1, \dots, n$, the minimum adequate radius is then

$$r_{i, \min} = 2 \max\{\|p_i - q\|_2 \mid q \in W(p_i, r_{i, \min})\}.$$

The minimum adequate radius across the overall network is then $r_{\min} = \max_{i \in \{1, \dots, n\}} r_{i, \min}$. The algorithm presented in Cortés et al. (2004) builds on these observations to compute the Voronoi partition of a bounded set generated by a pointset in a distributed way. •

2.2.2 The locally cliqueless graph of a proximity graph

Given a proximity graph, it is sometimes useful to construct another proximity graph that has fewer edges

and the same number of connected components. This is certainly the case when optimizing multi-agent cost functions in which the proximity graph edges describe pairwise constraints between agents. Additionally, the construction of the new proximity graph should be spatially distributed over the original proximity graph. Here, we present the notion of locally cliqueless graph of a proximity graph.

Let G be a proximity graph in the Euclidean space. The *locally cliqueless graph* $G_{lc,G}$ of G is the proximity graph defined by: $(p_i, p_j) \in E_{G_{lc,G}}(\mathbf{P})$ if

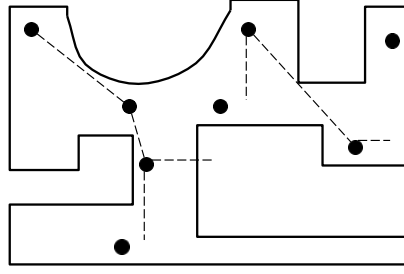


Figure 2.12 Locally cliqueless graph $G_{lc,G_{vis,Q}}$ of the visibility graph $G_{vis,Q}$ for the node configuration shown in Figure 2.9(d).

$(p_i, p_j) \in E_G(P)$ and

$$(p_i, p_j) \in E_{G_{EMST}}(P'),$$

for any maximal clique P' of (p_i, p_j) in P' . Figure 2.12 shows an illustration of this notion. The properties of this construction are summarized in the following result; for the proof, see [Ganguli et al. \(2009\)](#).

Theorem 2.11 (Properties of the locally cliqueless graph). *Let G be a proximity graph in the Euclidean space. Then, the following statements hold:*

- (i) $G_{EMST,G} \subseteq G_{lc,G} \subseteq G$;
- (ii) $G_{lc,G}$ has the same connected components as G ; and
- (iii) for $G = G_{disk}(r)$, $G_{vis,Q}$, and $G_{vis-disk,Q}$, where $r \in \mathbb{R}_{>0}$ and Q is an allowable environment, $G_{lc,G}$ is spatially distributed over G .

In general, the inclusions in Theorem 2.11(i) are strict.

2.2.3 Proximity graphs over tuples of points

The notion of proximity graph is defined for sets of distinct points $P =$

$\{p_1, \dots, p_n\}$. However, we will be interested in considering tuples of elements of X of the form $P = (p_1, \dots, p_n)$, where p_i corresponds to the position of an agent i of a robotic network. In principle, note that the tuple P might

contain coincident points. In order to reconcile this mismatch between sets and tuples, we will do the following.

Let $i_F : X^n \rightarrow F(X)$ be the natural immersion of X^n into $F(X)$, that is, $i_F(P)$ is the point set that contains only the distinct points in $P = (p_1, \dots, p_n)$. Note that i_F is invariant under permutations of its arguments

and that the cardinality of $i_F(p_1, \dots, p_n)$ is in general less than or equal to n . In what follows, $\mathbf{F} = i_F(P)$ will always denote the point set associated to $P \in X^n$. Using the natural inclusion, the notion of proximity graphs can be naturally extended as follows: given, we define (with a slight abuse of notation)

$$\mathbf{G} = \mathbf{G} \circ i_F : X^n \rightarrow \mathbf{G}(X).$$

Additionally, we define the set of neighbors map $\mathbf{N}_G : X \times X^n \rightarrow \mathbf{F}(X)$ by

$$\mathbf{N}_G(p, (p_1, \dots, p_n)) = \mathbf{N}_G(p, i_F(p_1, \dots, p_n)).$$

According to this definition, coincident points in the tuple $(p_1, \dots, p_n)$ will have the same set of neighbors. As before, it is convenient to define the shorthand notation $\mathbf{N}_{G,p} : X^n \rightarrow \mathbf{F}(X)$, $\mathbf{N}_{G,p}(P) = \mathbf{N}_G(p, P)$ for $p \in X$.

2.2.4 Spatially distributed maps

Given a set Y and a proximity graph $\mathbf{G}$, a map $T : X^n \rightarrow Y^n$ is *spatially distributed over $\mathbf{G}$* if there exists a map $\tilde{T} : X \times \mathbf{F}(X) \rightarrow Y$, with the property that, for all $(p_1, \dots, p_n) \in X^n$ and for all $j \in \{1, \dots, n\}$,

$$T_j(p_1, \dots, p_n) = \tilde{T}(p_j, \mathbf{N}_{G,p_j}(p_1, \dots, p_n)),$$

where T_j denotes the j th component of T . In other words, the j th component of a spatially distributed map at $(p_1, \dots, p_n)$ can be computed with only knowledge of the vertex p_j and the neighboring vertices in the undirected graph $\mathbf{G}(P)$.

When studying coordination tasks and coordination algorithms, it will be relevant to characterize the spatially distributed features of functions, vector fields, and set-valued maps with respect to suitable proximity graphs.

Remark 2.12 (Relationship with the notion of

spatially distributed graphs). Note that the proximity graph G_1 is spatially distributed over the proximity graph G_2 if and only if the map

$$P \in X^n \mapsto (N_{G_1}(P), \dots, N_{G_2}(P)) \in F(X)^n$$

is spatially distributed over G_2 . •

2.3 GEOMETRIC OPTIMIZATION PROBLEMS AND MULTICENTER FUNCTIONS

In this section we consider various interesting geometric optimization problems. By geometric optimization, we mean an optimization problem induced

by a collection of geometric objects (see [Boltyanski et al., 1999](#)). We shall pay particular attention to facility location problems, in which service sites are spatially allocated to fulfill a particular request.

2.3.1 Expected-value multicenter functions

Let $S \subset \mathbb{R}^d$ be a bounded environment of interest, and consider a density function $\varphi : \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$. For the discussion of this section, only the value of φ restricted to S is of interest. One can regard φ as a function measuring the probability that some event takes place over the environment. The larger the value of $\varphi(q)$, the more important the location q is. We refer to a non-increasing and piecewise continuously differentiable function $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$, possibly with finite jump discontinuities, as a *performance*. Performance functions describe the utility of placing a node at a certain distance from a location in the environment. The smaller the distance, the larger the value of f , that is, the better the performance. For instance, in servicing problems, performance functions can encode the travel time or the energy expenditure required to service a specific destination. In sensing problems, performance functions can encode the signal-to-noise ratio between a source with an unknown location and a sensor attempting to locate it.

Given a bounded measurable set $S \subset \mathbb{R}^d$, a density function φ , and a performance function f , let us consider the expected value of the coverage over any point in S provided by a set of points $p_1, \dots, p_n$. Formally, we define the *expected-value multicenter* function $H_{\text{exp}} : S^n \rightarrow \mathbb{R}$ by

$$H_{\text{exp}}(p_1, \dots, p_n) = \int_S \max_i \{f(\|q - p_i\|)\} \varphi(q) dq. \quad (2.3.1)$$

The definition of

H_{exp} can be interpreted as follows: for each location $q \in S$, consider the best coverage of q among those provided by each of the nodes $p_1, \dots, p_n$, which corresponds to the value $\max_{i \in \{1, \dots, n\}} f(q, p_i)$. Then, evaluate the performance by the importance $\varphi(q)$ of the location q . Finally, sum the resulting quantity over all the locations of the environment S , to obtain $H_{\text{exp}}(p_1, \dots, p_n)$ as a measure of the overall coverage provided by $p_1, \dots, p_n$.

Given the meaning of H_{exp} , we seek to solve the following geometric optimization problem:

$$\text{maximize } H_{\text{exp}}(p_1, \dots, p_n), \quad (2.3.2)$$

that is, we seek to determine a set of configurations $p_1, \dots, p_n$ that maximize the value of the multicenter function H_{exp} . An equivalent formulation of this problem is referred to as a *continuous p -median problem* in the literature

on facility location (see, e.g., [Drezner, 1995](#)). In our discussion, we will pay special attention to the case when $n = 1$, which we term the 1-center problem. For the purpose of solving (2.3.2), note that we can assume that the performance function satisfies $f(0) = 0$. This can be done without loss of generality, since for any $c \in \mathbb{R}$, one has

$$\int \max_{i \in \{1, \dots, n\}} (f(\|q - p_i\|) + c) \phi(q) dq = H_{\text{exp}}(p_1, \dots, p_n) + c A_\phi(S).$$

The expected-value multicenter function can be alternatively described in terms of the Voronoi partition of S generated by $p_1, \dots, p_n$. Let us define the set

$$S_{\text{coinc}} = \{(p_1, \dots, p_n) \in (\mathbb{R}^d)^n \mid p_i = p_j \text{ for some } i \neq j\},$$

consisting of tuples of n points, where some of them are repeated. Then, for $(p_1, \dots, p_n) \in S^n \setminus S_{\text{coinc}}$, one has

$$H_{\text{exp}}(p_1, \dots, p_n) = \sum_{i=1}^n \int_{V_i(P)} f(\|q - p_i\|) \phi(q) dq. \quad (2.3.3)$$

This expression of H_{exp} is appealing because it clearly shows the result of the overall coverage of the environment as the aggregate contribution of all individual nodes. If $(p_1, \dots, p_n) \in S_{\text{coinc}}$, then a similar decomposition of H_{exp} can be written in terms of the distinct points $P = \{p_1, \dots, p_n\}$.

Inspired by the expression (2.3.3), let us define a more general version

of the expected-value multicenter function. Given $(p_1, \dots, p_n) \in S^n$ and a partition $\{W_1, \dots, W_n\} \subset \mathcal{P}(S)$ of S , let

$$H_{\text{exp}}(p_1, \dots, p_n, W_1, \dots, W_n) = \sum_{i=1}^n \int_{W_i} f(\|q - p_i\|) \phi(q) dq. \quad (2.3.4)$$

Notice that $H_{\exp}(p_1, \dots, p_n) = \exp(p_1, \dots, p_n, V_1(\cdot), \dots, V_n(\cdot))$, for all $(p_1, \dots, p_n) \in S^n$. Moreover, one can establish the following optimality result (see Du et al., 1999).

Proposition 2.13 ($H_{\exp}$ -optimality of the Voronoi partition). *Let $P = \{p_1, \dots, p_n\} \in F(S)$. For any performance function f and for any partition $\{W_1, \dots, W_n\} \subset P(S)$ of S ,*

$$H_{\exp}(p_1, \dots, p_n, V_1(P), \dots, V_n(P)) \geq H_{\exp}(p_1, \dots, p_n, W_1, \dots, W_n),$$

and the inequality is strict if any set in $\{W_1, \dots, W_n\}$ differs from the corresponding set in $\{V_1(P), \dots, V_n(P)\}$ by a set of positive measure. In other words, the Voronoi partition $V(P)$ is optimal for $\exp$ among all partitions of S .

Proof. Assume that, for $i \neq j \in \{1, \dots, n\}$, the set $\text{int}(W_i) \cap \text{int}(V_j(\mathbf{P}))$ has strictly positive measure. For all $q \in \text{int}(W_i) \cap \text{int}(V_j(\mathbf{P}))$, we know that $\|q - p_i\|^2 > \|q - p_j\|^2$. Because f is non-increasing, $f(\|q - p_i\|^2) < f(\|q - p_j\|^2)$ and, since $\text{int}(W_i) \cap \text{int}(V_j(\mathbf{P}))$ has strictly positive measure,

$$\int_{\text{int}(W_i) \cap \text{int}(V_j(\mathbf{P}))} f(\|q - p_i\|^2) \phi(q) dq < \int_{\text{int}(W_i) \cap \text{int}(V_j(\mathbf{P}))} f(\|q - p_j\|^2) \phi(q) dq.$$

Therefore, we deduce

$$\int_{W_i} f(\|q - p_i\|^2) \phi(q) dq < \sum_{j=1}^n \int_{W_i \cap V_j(\mathbf{P})} f(\|q - p_j\|^2) \phi(q) dq,$$

and the statements follow. $\square$

Different performance functions lead to different expected-value multicenter functions. Let us examine some important cases.

Distortion problem: Consider the performance function $f(x) = x^2$. Then, on S^n_{coinc} , the expected-value multicenter function takes the form

$$H_{\text{dist}}(p_1, \dots, p_n) = - \sum_{i=1}^n \int_{V_i(\mathbf{P})} \|q - p_i\|^2 \phi(q) dq = - \sum_{i=1}^n J_\phi(V_i(\mathbf{P}), p_i),$$

where recall that $J_\phi(W, p)$ denotes the polar moment of inertia of the set W about the point p . In signal compression dist is referred to $\square$

as the *distortion function* and is relevant in many disciplines including vector quantization, signal compression, and numerical integration (see [Gray and Neuhoﬀ, 1998](#); [Du et al., 1999](#)). Here, distortion refers to the average deformation (weighted by the density ϕ) caused by reproduc-

ing $q \in S$ with the location p_i in $\mathbb{P} \{p_1, \dots, p_n\}$ such that $q \in V_i$. It is interesting to note that

$$\begin{aligned} H_{\text{dist}}(p_1, \dots, p_n, W_1, \dots, W_n) &= - \sum_{i=1}^n J_{\phi}(W_i, p_i) \\ &= - \sum_{i=1}^n J_{\phi}(W_i, \text{CM}_{\phi}(W_i)) - \sum_{i=1}^n A_{\phi}(W_i) \|p_i - \text{CM}_{\phi}(W_i)\|_2^2 \quad (2.3.5) \end{aligned}$$

where in the last equality we have used the Parallel Axis Theorem (Hibbeler, 2006). Note that the first term only depends on the partition of S , whereas the second term also depends on the location of the points. The following result is a consequence of this observation.

Proposition 2.14 (H_{dist} -optimality of centroid locations). *Let*

$\{W_1, \dots, W_n\} \subset \mathcal{P}(S)$ *be a partition of* S . *Then, for any set points*

$$P = \{p_1, \dots, p_n\} \in \mathcal{F}(S),$$

$$H_{\text{dist}}(\text{CM}_\phi(W_1), \dots, \text{CM}_\phi(W_n), W_1, \dots, W_n) \geq H_{\text{dist}}(p_1, \dots, p_n, W_1, \dots, W_n),$$

and the inequality is strict if there exists $i \in \{1, \dots, n\}$ *for which* W_i *has non-vanishing area and* $p_i \neq \text{CM}_\phi(W_i)$. *In other words, the centroid locations* $\text{CM}_\phi(W_1), \dots, \text{CM}_\phi(W_n)$ *are optimal for* H_{dist} *among all configurations in* S .

A consequence of this result is that for the 1-center problem, that is, when $n = 1$, the node location that optimizes $p \mapsto H_{\text{dist}}(p) =$

$J_\phi(S, p)$ is the centroid of the set S , denoted by $\text{CM}_\phi(S)$.

Area problem: For $a \in \mathbb{R}_{>0}$, consider the performance

function $f(x) = 1_{[0,a]}(x)$, that is, the indicator function of the closed interval $[0, a]$.

Then, the expected-value multicenter function takes the form

$$\begin{aligned} H_{\text{area},a}(p_1, \dots, p_n) &= \sum_{i=1}^n \int_{V_i(P)} 1_{[0,a]}(|q - p_i|) \phi(q) dq \\ &= \sum_{i=1}^n \int_{V_i(P) \cap B(p_i, a)} \phi(q) dq \\ &= \sum_{i=1}^n A_\phi(V_i(P) \cap \overline{B(p_i, a)}) = A_\phi(\cup_{i=1}^n \overline{B(p_i, a)}), \end{aligned}$$

that is, it corresponds to the area, measured according to ϕ , covered by the union of the n balls $B(p_1, a), \dots, B(p_n, a)$. Exercise E2.5 discusses the 1-center area problem.

Mixed distortion-area problem: For $a \in \mathbb{R}_{>0}$ and $b \leq$

$-a^2$, consider the performance function $f(x) = -x^2 1_{[0,a]}(x) + b \cdot 1_{]a,+\infty[}(x)$. Then, on $S^n \setminus S_{\text{coinc}}$, the expected-value multicenter function takes the form

$$H_{\text{dist-area},a,b}(p_1, \dots, p_n) = - \sum_{i=1}^n J_\phi(V_{i,a}(P), p_i) + b A_\phi(Q \setminus \bigcup_{i=1}^n B(p_i, a)),$$

that is, it is a combination of the multicenter functions corresponding to the distortion problem and the area problem. Of special interest to us is the multicenter function that results from the choice $b = a^2$.

In this case, the performance function f is continuous, and we simply write $H_{\text{dist-area},a}$. The extension of this function to sets of points and

partitions of the space reads as follows:

$$\begin{aligned} H_{\text{dist-area},a}(p_1, \dots, p_n, W_1, \dots, W_n) \\ = \sum_{i=1}^n \left(J_\phi(W_i \cap B(p_i, a), p_i) + a^2 A_\phi(W_i \cap (S \setminus B(p_i, a))) \right). \end{aligned}$$

We leave the proof of the following optimality result as a guided exercise for the reader (see Exercise E2.10).

Proposition 2.15 ($H_{\text{dist-area},a}$ -optimality of centroid locations).

Let $\{W_1, \dots, W_n\} \subset \mathcal{P}(S)$ be a partition of S . Then, for any $P =$

$\{p_1, \dots, p_n\} \in F(S)$,

$$\begin{aligned} H_{\text{dist-area},a}(q_1^*, \dots, q_n^*, W_1, \dots, W_n) \\ \geq H_{\text{dist-area},a}(p_1, \dots, p_n, W_1, \dots, W_n), \end{aligned}$$

where we have used the shorthands $q_i^* = \text{CM}_\phi(W_i \cap \overline{B(p_i, a)})$, for $i \in \{1, \dots, n\}$. Furthermore, the inequality is strict if there exists $i \in \{1, \dots, n\}$ for which W_i has non-vanishing area and $p_i \neq q_i^*$.

A consequence of this result is that for the 1-center problem, that is, when $n = 1$ —the node location that optimizes $p \mapsto H_{\text{dist-area},a}(p) = J_\phi(S \cap B(p, a), p) + a^2 A_\phi(S \setminus \overline{B(p, a)})$ is the centroid of the set $S \cap B(p, a)$, denoted by $\text{CM}_\phi(S \cap B(p, a))$.

Next, we characterize the smoothness of the expected-value multicenter function. Before stating the precise result, let us introduce some useful notation. For a performance function f , let $\text{Dscn}(f)$ denote the (finite) set of points where f is discontinuous. For each $a \in \text{Dscn}(f)$, define the limiting values from the left and from the right, respectively, as

$$f_-(a) = \lim_{x \rightarrow a^-} f(x), \quad f_+(a) = \lim_{x \rightarrow a^+} f(x).$$

We are now ready to characterize the smoothness of $\mathbf{H}_{\text{exp}}$, whose proof is given in Section 2.5.3. Before stating the result, recall that the line integral of a function $g : \mathbb{R}^2 \rightarrow \mathbb{R}$ over a curve C parameterized by a continuous and piecewise continuously differentiable map $\gamma : [0, 1] \rightarrow \mathbb{R}^2$ is defined by

$$\int_C g = \int_C g(\gamma) d\gamma := \int_0^1 g(\gamma(t)) \|\dot{\gamma}(t)\|_2 dt,$$

and is independent of the selected parameterization.

Theorem 2.16 (Smoothness properties of $\mathbf{H}_{\text{exp}}$). *Given a set $S \subset \mathbb{R}^d$ that is bounded and measurable, a density $\varphi : \mathbb{R} \rightarrow \mathbb{R}_{\geq 0}$, and a performance function $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$, the expected-value multicenter function $\mathbf{H} : S^n \rightarrow \mathbb{R}$ is*

- (i) globally Lipschitz² on S^n ; and
(ii) continuously differentiable on $S^n \setminus S_{\text{coinc}}$, where for $i \in \{1, \dots, n\}$

$$\frac{\partial H_{\text{exp}}(P)}{\partial p_i} = \int_{V_i(P)} \frac{\partial}{\partial p_i} f(\|q - p\|^2) \phi(q) dq + \sum_{a \in \text{Dscn}(f)} [f_-(a) - f_+(a)] \sum \int_{V_i(P) \cap \partial B(p_i, a)} n_{\text{out}}(q) \phi(q) dq, \quad (2.3.6)$$

where n_{out} is the outward normal vector to $B(p_i, a)$. Therefore, the gradient of H_{exp} , interpreted as a map from S^n to $\mathbb{R}^n$, is spatially distributed (in the sense defined in Section 2.2.4) over the Delaunay graph G_D .

Let us discuss how Theorem 2.16 particularizes to the distortion, area, and mixed distortion-area problems.

Distortion problem: In this case, the performance function does not have any discontinuities and, therefore, the second term in (2.3.6) vanishes. The gradient of H_{dist} on $S^n \setminus S_{\text{coinc}}$ then takes the form, for each

$i \in \{1, \dots, n\}$,

$$\frac{\partial H_{\text{dist}}(P)}{\partial p_i} = 2 A(V(P)) (CM(V(P)) - p_i),$$

that is, the i th component of the gradient points in the direction of the vector going from p_i to the centroid of its Voronoi cell. The critical points of H_{dist} are therefore the set of centroidal Voronoi configurations

in S (cf. Section 2.1.4). This is a natural generalization of the result for the 1-center case, where the optimal node location is the centroid $CM_\phi(S)$.

Area problem: In this case, the performance function is differentiable everywhere except at a single

discontinuity, and its derivative is identically zero. Therefore, the first term in (2.3.6) vanishes. The gradient of $H_{\text{area},a}$ on $S^n \setminus S_{\text{coinc}}$ then takes the form,

$$\frac{\partial H_{\text{area},a}}{\partial p_i}(P) = \int_{V_i(P) \cap \partial B(p_i, a)} \langle n_{\text{out}}(q), \varphi(q) \rangle dq,$$

for each $i \in \{1, \dots, n\}$,

where n_{out} is the outward normal vector to $B(p_i, a)$. The integrand is an average of the normal at each point of $V_i(P) \cap \partial B(p_i, a)$, as illustrated in Figure 2.13. The critical points of $H_{\text{area},a}$ correspond to configurations with the property that each p_i is a local maximum for the area of $V_{i,a}(P) = V_i(P) \cap B(p_i, a)$ at fixed $V_i(P)$. We refer to these

²Given $S \subset \mathbb{R}^h$, a function $f: S \rightarrow \mathbb{R}^k$ is globally Lipschitz if there exists $K \in \mathbb{R}_{>0}$ such that

$\|f(x) - f(y)\|_2 \leq K \|x - y\|_2$ for all $x, y \in S$.

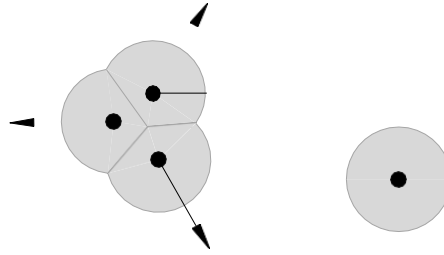


Figure 2.13 The gradient of the area function when the density function is constant. The component of the gradient corresponding to the rightmost node is zero; there is no incentive for this node to move in any particular direction. The component of the gradient for each of the three leftmost agents is non-zero; roughly speaking, by moving along the gradient directions, these agents decrease the overlapping among their respective disk and cover new regions of the space.

configurations as *a-limited area-centered* Voronoi configurations. This is a natural generalization of the result for the 1-center case, where the optimal node location maximizes $A_\phi(S \cap B(p, a))$ (cf., Exercise E2.5).

Mixed distortion-area problem: In this case, the gradient of the multi-center function $H_{\text{dist-area},a,b}$ is a combination of the gradients of H_{dist} and $H_{\text{area},a}$. Specifically, one has for each $i \in \{1, \dots, n\}$,

$$\frac{\partial H_{\text{dist-area},a,b}}{\partial p_i}(P) = 2A(V_{\phi_i,a}(P))(\text{CM}_{\phi_i,a}(P)) - p_i - (a^2 + b) \int_{V_i(P) \cap \partial \overline{B(p_i,a)}} n_{\text{out}}(q) \phi(q) dq,$$

where n_{out} is the outward normal vector to $B(p_i, a)$. For the particular case when $b = -a^2$, the performance function is continuous, and the gradient of $H_{\text{dist-area},a}$ takes the simpler form

$$\frac{\partial H_{\text{dist-area},a}}{\partial p_i}(P) = 2A(V_{\phi_i,a}(P))(\text{CM}_{\phi_i,a}(P)) - p_i,$$

which points in the direction of the vector from p_i to the centroid of its a -limited Voronoi cell. In this case, the critical points of dist-area_a are therefore the set of a -limited centroidal Voronoi configurations in S (cf., Section 2.1.4). This is a natural generalization of the result for the 1-center case, where the optimal node location is the centroid $\text{CM}_\phi(S \cap B(p, a))$.

We refer to dist , area_a , and dist-area_a as multicenter functions because, as the above discussion shows, their critical points correspond to various notions of center Voronoi configurations.

Note that the gradients of $H_{\text{area},a}$ and $H_{\text{dist-area},a,b}$ are spatially distributed over the $2a$ -limited Delaunay graph $\text{LD}(2a)$. This observation is important for practical considerations: robotic agents with range-limited interactions cannot in general compute the gradient of H_{dist} because, as we noted in Remark 2.10, for a given $r \in \mathbb{R}_{>0}$, G_D is not in general spatially distributed over $G_{\text{disk}}(r)$. However, robotic agents with range-limited interactions can compute the gradients of $H_{\text{area},a}$ and $H_{\text{dist-area},a,b}$ as long as $r \geq 2a$ because, from Theorem 2.7(iii), $\text{LD}(r)$ is spatially distributed over $\text{disk}(r)$. The relevance of this fact is further justified by the following result.

Proposition 2.17 (Constant-factor approximation of H_{dist}). *Let $S \subset \mathbb{R}^d$ be bounded and measurable. Consider the mixed distortion-area problem with $a \in]0, \text{diam}(S)]$ and $b = -\text{diam}(S)^2$. Then, for all $P \in S^n$,*

$$H_{\text{dist-area},a,b}(P) \leq H_{\text{dist}}(P) \leq \beta H_{\text{dist-area},a,b}(P) < 0, \quad (2.3.7)$$

where $\beta = \frac{a}{\text{diam}(S)} \in [0, 1]$.

In fact, similar constant-factor approximations of the expected-value multicenter function H_{exp} can also be established (see Cortés et al., 2005).

2.3.2 Worst-case and disk-covering multicenter functions

Given a compact set $S \subset \mathbb{R}^d$ and a performance function f , let us consider the point in S that is worst covered by a set of points $p_1, \dots, p_n$. Formally, we define the *worst-case multicenter* function $H_{\text{worst}} : S^n \rightarrow \mathbb{R}$ by

$$H_{\text{worst}}(p_1, \dots, p_n) = \min_{q \in S} \max_{i \in \{1, \dots, n\}} f(\|q - p_i\|_2). \quad (2.3.8)$$

The definition of H_{worst} can be read as follows: for each location $q \in S$, consider the best coverage of q among those

provided by each of the nodes $p_1, \dots, p_n$, which corresponds to the value $\max_{i \in \{1, \dots, n\}} f(q - p_i)$. Then, compute the worst coverage $H_{\text{worst}}(p_1, \dots, p_n)$ by comparing the performance at all locations in S .

Given the interpretation of H_{worst} , we seek to solve the following geometric optimization problem:

$$\text{maximize } H_{\text{worst}}(p_1, \dots, p_n), \quad (2.3.9)$$

that is, we seek to determine configurations $p_1, \dots, p_n$ that maximize the value of H_{worst} . An equivalent formulation of this problem is referred to as a *continuous p -center problem* in the literature on facility location (see, e.g., [Drezner, 1995](#)).

In the present context, also relevant is the *disk-covering multicenter* function $H_{dc} : S^n \rightarrow \mathbb{R}$, defined by

$$H_{dc}(p_1, \dots, p_n) = \max_{q \in S} \min_{i \in \{1, \dots, n\}} \|q - p_i\|. \quad (2.3.10)$$

The value of H_{dc} can be interpreted as the largest possible distance from a point in S to one of the locations $p_1, \dots, p_n$. Note that, by definition, the environment S is contained in the union of n closed balls centered at $p_1, \dots, p_n$ with radius $H_{dc}(p_1, \dots, p_n)$. The definition of H_{dc} is illustrated in Figure 2.14(a).

The following result establishes the relationship between the worst-case and the disk-covering multicenter functions, and as byproduct, provides an elegant reformulation of the geometric optimization problem (2.3.9). Its proof is left to the reader.

Lemma 2.18 (Relationship between H_{worst} and H_{dc}). *Given $S \subset \mathbb{R}^d$ compact and a performance function $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$, one has $H_{\text{worst}} = f \circ H_{dc}$.*

Using Lemma 2.18 and the fact that f is non-increasing, we can reformulate the geometric optimization problem (2.3.9) as

$$\text{minimize } H_{dc}(p_1, \dots, p_n), \quad (2.3.1)$$

1) that is, find the minimum radius r such that the environment S is covered by n closed balls centered at $p_1, \dots, p_n$ with equal radius r . Note the connection between this formulation and the classical disk-covering problem: how to cover a region with (possibly overlapping) disks of minimum radius. We shall comment more on this connection later.

Because of the equivalence between the geometric

optimization problems (2.3.9) and (2.3.11), we focus our attention on H_{dc} . The disk-covering multicenter function can be alternatively described in terms of the Voronoi

partition of S generated by $P = \{p_1, \dots, p_n\}$. For $(p_1, \dots, p_n) \in S^n \setminus S_{\text{coinc}}$,

$$\begin{aligned} H_{dc}(p_1, \dots, p_n) &= \max_{i \in \{1, \dots, n\}} \max_{\{q \in V_i(P)\}} \|q - p_i\|^2 \\ &= \max_{i \in \{1, \dots, n\}} \max_{q \in \partial V_i(P)} \|q - p_i\|^2. \end{aligned} \quad (2.3.12)$$

This characterization of H_{dc} is illustrated in Figure 2.14(b). The expression (2.3.12) is appealing because it clearly shows the value of H_{dc} as the result of the aggregate contribution of all individual nodes. If $(p_1, \dots, p_n) \in S_{\text{coinc}}$, then a similar decomposition of H_{dc} can be written in terms of the distinct points $P = i_F(p_1, \dots, p_n)$. A node $i \in \{1, \dots, n\}$ is called *active* at $(p_1, \dots, p_n)$ if $\max_{q \in \partial V_i(P)} \|q - p_i\|^2 = H_{dc}(p_1, \dots, p_n)$. A node is *passive* at $(p_1, \dots, p_n)$ if it is not active.

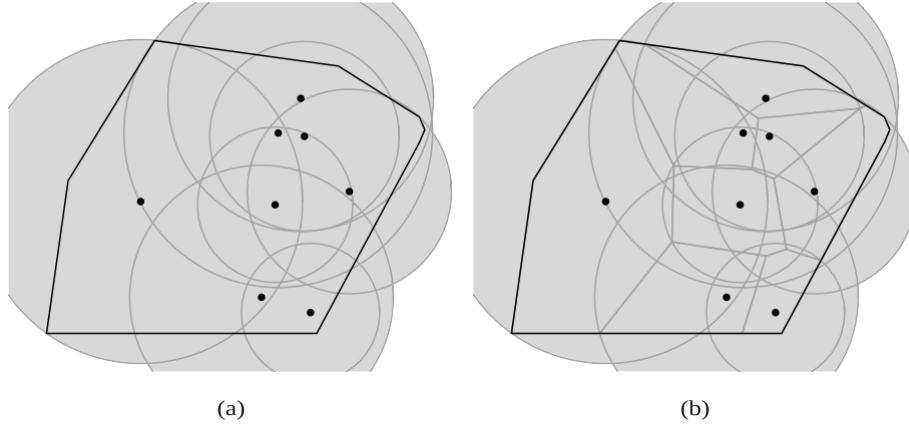


Figure 2.14 An illustration of the definition of H_{dc} : (a) and (b) show the same configuration, with and without the Voronoi configuration, respectively. For each node, the disk is the minimum-radius disk centered at the node and enclosing the Voronoi cell. The value of H_{dc} is the radius of the disk centered at the leftmost node.

Inspired by expression (2.3.12), let us define a more general version of the disk-covering multicenter function. Given $(p_1, \dots, p_n) \in S^n$ and a partition $\{W_1, \dots, W_n\} \subset \mathcal{P}(S)$ of S , let

$$H_{dc}(p_1, \dots, p_n, W_1, \dots, W_n) = \max_{i \in \{1, \dots, n\}} \max_{q \in \partial W_i} \|q - p_i\|.$$

Note the relationship $H_{dc}(p_1, \dots, p_n) = H_{dc}(p_1, \dots, p_n, V_1(P), \dots, V_n(P))$, for all $(p_1, \dots, p_n) \in S^n$. Moreover, one can establish the following optimality result, whose proof is given in Section 2.5.4.

Proposition 2.19 (H_{dc} -optimality of the Voronoi partition and circumcenter locations). *For any $P = \{p_1, \dots, p_n\} \in \mathcal{F}(S)$ and any partition $\{W_1, \dots, W_n\} \subset \mathcal{P}(S)$ of S ,*

$$H_{dc}(p_1, \dots, p_n, V_1(P), \dots, V_n(P)) \leq H_{dc}(p_1, \dots, p_n, W_1, \dots, W_n),$$

that is, the Voronoi partition $V(P)$ is optimal for H_{dc} among all partitions of S , and

$$H_{dc}(CC(W_1), \dots, CC(W_n), W_1, \dots, W_n) \leq H_{dc}(p_1, \dots, p_n, W_1, \dots, W_n),$$

that is, the circumcenter locations $CC(W_1), \dots, CC(W_n)$ are optimal for H_{dc}

among all configurations in S .

As a corollary of this result, we have that the circumcenter of S is a global optimum of H_{dc} for the 1-center problem, that is, when $n = 1$. This comes as no surprise since, in this case, the value $_{dc}(p)$ corresponds to the radius of the minimum-radius sphere centered at p that encloses S .

The following result characterizes the smoothness properties of the disk-covering multicenter function; for more details and for the proof, see Cortés and Bullo (2005).

Theorem 2.20 (Smoothness properties of H_{dc}). *Given $S \subset \mathbb{R}^d$ compact, the disk-covering multicenter function $H_{dc} : \mathbb{R}^n \rightarrow \mathbb{R}$ is globally Lipschitz on S^n .*

The generalized gradient and the critical points of H_{dc} can be characterized, but require a careful study based on nonsmooth analysis (Clarke, 1983). In particular, two facts taken from Cortés and Bullo (2005) are of interest here. First, under certain technical conditions, one can show that the critical points of H_{dc} are circumcenter Voronoi configurations. This is why we refer to H_{dc} as a multicenter function. Second, the generalized gradient of H_{dc} is not spatially distributed over D . This is essentially due to the inherent comparison among all agents that is embedded in the definition of H_{dc} (via the max function).

2.3.3 Sphere-packing multicenter functions

Given a compact connected set $S \subset \mathbb{R}^d$, imagine trying to fit inside S “maximally large” non-intersecting balls. Assuming that the balls are centered at a set of points $p_1, \dots, p_n$, we aim to maximize their smallest radius. We define the *sphere-packing multicenter* function $H_{sp} : S^n \rightarrow \mathbb{R}$ by

$$H_{sp}(p_1, \dots, p_n) = \min_{j \in \{1, \dots, n\}} \frac{1}{2} \min \left\{ \|p_i - p_j\|_2, \text{dist}(p_i, \partial S) \right\}. \quad (2.3.13)$$

The definition of H_{sp} can be read as follows: consider the pairwise distances between any two points p_i, p_j (multiplied by a factor $1/2$ so that each point can fit a ball of equal radius and these balls do not intersect), and the indi-

vidual distances from each point to the boundary of the environment. The value of H_{sp} is then the smallest of all distances, guaranteeing that the union of n open balls centered at $p_1, \dots, p_n$ with radius $H_{sp}(p_1, \dots, p_n)$ is disjoint and contained in S . The definition of H_{sp} is illustrated in Figure 2.15(a).

Given the definition of H_{sp} , we seek to solve the following geometric optimization problem:

$$\text{maximize } H_{sp}(p_1, \dots, p_n), \quad (2.3.14)$$

that is, we seek to determine configurations $p_1, \dots, p_n$ that maximize the value of H_{sp} . Note the connection of this formulation with the classical sphere-packing problem: how to maximize the number of fixed-radius non-overlapping spheres inside a region.

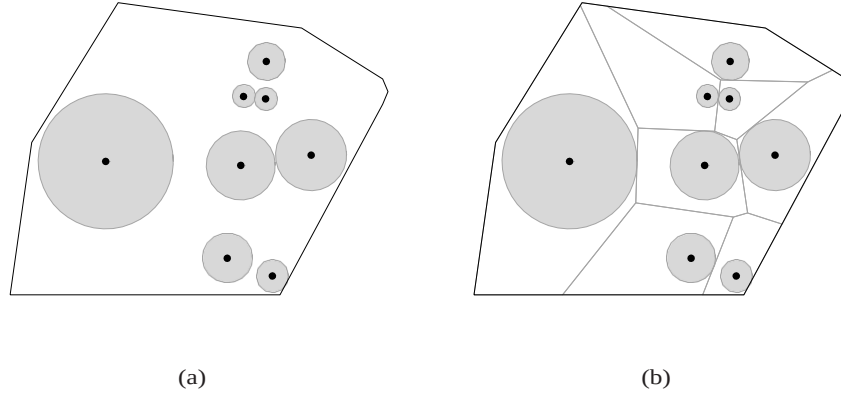


Figure 2.15 An illustration of the definition of H_{sp} : (a) and (b) show the same configuration, with and without the Voronoi configuration, respectively. For each node, the disk is the maximum-radius disk centered at the node and contained in the Voronoi cell. The value of H_{sp} is the radius of the two equal-radius smallest disks.

The sphere-packing multicenter function can be alternatively described in terms of the Voronoi partition of S generated by $P = \{p_1, \dots, p_n\}$. For

$(p_1, \dots, p_n) \in S^n \setminus S_{\text{coinc}}$, one has

$$H_{sp}(p_1, \dots, p_n) = \min_{i \in \{1, \dots, n\}} \min_{q \in \partial V_i(P)} \|q - p_i\|. \quad (2.3.15)$$

This description is illustrated in Figure 2.15(b). As for the previous multi-center functions, expression (2.3.15) is appealing because it clearly shows the value of H_{sp} as the result of the aggregate contribution of all individual nodes. If $(p_1, \dots, p_n) \in S_{\text{coinc}}$, then a similar decomposition of H_{sp} exists in terms of the distinct points $P = i_F(p_1, \dots, p_n)$. A node $i \in \{1, \dots, n\}$ is called *active* at $(p_1, \dots, p_n)$ if $\min_{q \in \partial V_i(P)} \|q - p_i\| = H_{sp}(p_1, \dots, p_n)$. A node is *passive* at $(p_1, \dots, p_n)$ if it is not active.

Inspired by expression (2.3.15), let us define a more general version of

the sphere-packing multicenter function. Given $(p_1, \dots, p_n) \in S^n$ and a partition $\{W_1, \dots, W_n\} \subset P(S)$ of S , let

$$H_{\text{sp}}(p_1, \dots, p_n, W_1, \dots, W_n) = \min_{i \in \{1, \dots, n\}} \min_{q \in \partial W_i} \|q - p_i\|_2.$$

Note the relationship $H_{\text{sp}}(p_1, \dots, p_n) = H_{\text{sp}}(p_1, \dots, p_n, V_1(P), \dots, V_n(P))$, for all $(p_1, \dots, p_n) \in S^n \setminus S_{\text{coinc}}$. Additionally, note that the quantity $\text{sp}(q_1, \dots, q_n, W_1, \dots, W_n)$ is the same for any $q_i \in \text{IC}(W_i)$, $i = 1, \dots, n$. With a slight abuse of notation, we refer to this common value using the

symbol $H_{\text{sp}}(\text{IC}(W_1), \dots, \text{IC}(W_n), W_1, \dots, W_n)$. Moreover, one can establish the following optimality result (for the proof, see Section 2.5.5).

Proposition 2.21 (H_{sp} -optimality of the Voronoi partition and in-center locations). *For any $P = \{p_1, \dots, p_n\} \in F(S)$ and any partition*

$\{W_1, \dots, W_n\} \subset P(S)$ of S ,

$$H_{sp}(p_1, \dots, p_n, V_1(P), \dots, V_n(P)) \geq H_{sp}(p_1, \dots, p_n, W_1, \dots, W_n),$$

that is, the Voronoi partition $V(P)$ is optimal for $_{sp}$ among all partitions of S , and

$$H_{sp}(IC(W_1), \dots, IC(W_n), W_1, \dots, W_n) \geq H_{sp}(p_1, \dots, p_n, W_1, \dots, W_n),$$

that is, the incenters $IC(W_1), \dots, IC(W_n)$ are optimal for $_{sp}$ among all configurations in S .

As a corollary of this result, we have that the incenter set of S is composed of global optima of H_{sp} for the 1-center problem, that is, when $n = 1$. This comes as no surprise since, in this case, the value $_{sp}(p)$ corresponds to the radius of the maximum-radius sphere centered at p enclosed in S . The following result characterizes the smoothness properties of the sphere-packing multicenter function (see [Cortés and Bullo, 2005](#)).

Theorem 2.22 (Smoothness properties of $_{sp}$). *Given $S \subset \mathbb{R}^d$ compact, the sphere-packing multicenter function $_{sp} H^n$ on S^n is globally Lip-schitz on S^n .*

We conclude this section with some remarks that are analogous to those for the function H_{dc} . The generalized H_{sp} gradient and the critical points of $_{sp}$ can be characterized, but require a careful study based on nonsmooth analysis ([Clarke, 1983](#)). In particular, two facts taken from [Cortés and Bullo \(2005\)](#) are of interest here. First, under certain technical conditions, one can show that the critical points of H_{sp} are incenter Voronoi configurations. This is why we refer to H_{sp} as a multicenter function. Second, the generalized gradient of $_{sp}$ is not spatially distributed over D . This is essentially due to the inherent comparison among all agents that is

embedded in the definition of H_{sp} (via the min function).

2.4 NOTES

A thorough introduction to computational geometric concepts can be found in [Preparata and Shamos \(1993\)](#), [de Berg et al. \(2000\)](#), and [O'Rourke \(2000\)](#). The handbooks [Goodman and O'Rourke \(2004\)](#) and [Sack and Urrutia \(2000\)](#) present a comprehensive overview of computational geometric problems and their applications. Among the numerous topics that we do not discuss in this chapter, we mention distance geometry and rigidity theory ([Whiteley, 1997](#)), which are notable for their applications to network localization and formation control.

The notion of Voronoi partition, and generalizations of it, have been applied in numerous areas, including spatial interpolation, pattern analysis, spatial processes modeling, and optimization, to name a few. The survey [Aurenhammer \(1991\)](#) and the book by [Okabe et al. \(2000\)](#) discuss the history, properties, and applications of Voronoi partitions. The nearest-neighbor and natural-neighbor interpolations based on Voronoi partitions (see, for example [Sibson, 1981](#); [Boissonnat and Cazals, 2002](#)) are of particular interest to the treatment of this chapter because of their spatially distributed computation character. Spatially distributed maps for motion coordination are discussed in [Martínez et al. \(2007c\)](#) and adopted in later chapters.

Proximity graphs ([Jaromczyk and Toussaint, 1992](#)) are a powerful tool to capture the structure and shape of geometric objects, and therefore have applications in multiple areas, including topology control of wireless networks ([Santi, 2005](#)), computer graphics ([Langetepe and Zachmann, 2006](#)), and geographic analysis ([Radke, 1988](#)). The connectivity properties of certain proximity graphs (including those stated in Theorem 2.8) are taken from [Cortés et al. \(2005, 2006\)](#). In cooperative control, a closely related notion is that of state-dependent graph ([Mesbahi, 2005](#)). Random geometric graphs ([Penrose, 2003](#)) and percolation theory ([Bollobás and Riordan, 2006](#); [Meester and Roy, 2008](#)) study the properties of proximity graphs associated to the random deployment of points according to some specified density function.

Locational optimization problems ([Drezner, 1995](#); [Drezner and Hamacher, 2001](#)) are spatial resource-allocation problems (e.g., where to place mail-boxes in a city, or where to place cache servers on the internet) that pervade a broad spectrum of scientific disciplines.

Computational geometry plays an important role in locational optimization (Robert and Toussaint, 1990; Okabe et al., 2000). The field of geometric optimization (Mitchell, 1997; Agarwal and Sharir, 1998; Boltyanski et al., 1999) blends the geometric and locational optimization aspects to study a wide variety of optimization problems induced by geometric objects. The smoothness properties of the cost function H_{exp} are taken from Cortés et al. (2005).

2.5 PROOFS

This section gathers the proofs of the main results presented in the chapter.

2.5.1 Proofs of Theorem 2.7 and Theorem 2.8

Proof of Theorem 2.7. The inclusions in fact (i) are taken from Jaromczyk and Toussaint (1992), and de Berg et al. (2000). The proof of the first inclusion in fact (ii) is as follows. Let $(p_i, p_j) \in E_{G_{\text{G}} \cap G_{\text{disk}}(r)}(\mathbf{P})$. From the definition of the Gabriel graph, we deduce that $\|p_i - p_j\|_2 = \frac{\|p_i - p_k\|_2 + \|p_k - p_l\|_2}{2}$ for all $k \in \{1, \dots, n\} \setminus \{i, j\}$, and therefore, $\frac{\|p_i - p_k\|_2 + \|p_k - p_l\|_2}{2} \in V_i(\mathbf{P}) \cap V_j(\mathbf{P})$. Since $(p_i, p_j) \in E_{G_{\text{disk}}(r)}(\mathbf{P})$, we deduce that $\frac{\|p_i - p_l\|_2}{2} \in B(p_i, \frac{r}{2}) \cap B(p_j, \frac{r}{2})$, and hence $(p_i, p_j) \in E_{G_{\text{LD}}(r)}(\mathbf{P})$. The second inclusion in (ii) is straightforward: if $(p_i, p_j) \in E_{G_{\text{LD}}(r)}(\mathbf{P})$, then $V_i(\mathbf{P}) \cap V_j(\mathbf{P}) \neq \emptyset$, that is, $(p_i, p_j) \in E_{G_{\text{D}}(\mathbf{P})}$. Since clearly $(p_i, p_j) \in E_{G_{\text{disk}}(r)}(\mathbf{P})$, we conclude (ii). $\square$

Proof of Theorem 2.8. The proof of fact (i) is as follows. Let $\mathbf{P} \in F(\mathbb{R}^d)$. If $G_{\text{EMST}}(\mathbf{P}) \subseteq G_{\text{disk}}(r)(\mathbf{P})$, then clearly $G_{\text{disk}}(r)(\mathbf{P})$ is connected. To prove the other implication, we reason by contradiction. Assume $G_{\text{disk}}(r)(\mathbf{P})$ is connected and let $G_{\text{EMST}}(\mathbf{P}) \not\subseteq G_{\text{disk}}(r)(\mathbf{P})$, that is, there exist p_i and p_j with $(p_i, p_j) \in E_{G_{\text{EMST}}}(\mathbf{P})$ and $\|p_i - p_j\|_2 > r$. If we remove this edge from $E_{G_{\text{EMST}}}(\mathbf{P})$, then the tree becomes disconnected into two connected components T_1 and T_2 , with $p_i \in T_1$ and $p_j \in T_2$. Now, since by hypothesis $G_{\text{disk}}(r)(\mathbf{P})$ is connected, there must exist $k, l \in \{1, \dots, n\}$ such that $p_k \in T_1, p_l \in T_2$ and $\|p_k - p_l\|_2 \leq r$. If we add the edge (p_k, p_l) to the set of edges of $T_1 \cup T_2$, then the resulting graph tt is acyclic, connected, and contains all the vertices $\mathbf{P}$, that is, tt is a spanning tree. Moreover, since $\|p_k - p_l\|_2 \leq r < \|p_i - p_j\|_2$ and T_1 and T_2 are induced subgraphs of $G_{\text{EMST}}(\mathbf{P})$, we conclude that tt has smaller length than $G_{\text{EMST}}(\mathbf{P})$, which is a contradiction with the definition of the Euclidean minimum spanning tree.

Next, we prove fact (ii). For $r \in \mathbb{R}_+$, it suffices for us to show that $G_{\text{EMST}} \cap G_{\text{disk}}(r)$ has the same connected components as $G_{\text{disk}}(r)$, since this implies that the same result holds for $G_{\text{RN}} \cap G_{\text{disk}}(r)$, $G_{\text{G}} \cap G_{\text{disk}}(r)$, and $G_{\text{LD}}(r)$. Since $G_{\text{EMST}} \cap G_{\text{disk}}(r)$ is a subgraph of $G_{\text{disk}}(r)$, it is clear that vertices belonging to the same connected component of $G_{\text{EMST}} \cap G_{\text{disk}}(r)$ must also belong to the same connected component of $G_{\text{disk}}(r)$. To prove the converse, let $P \in F(\mathbb{R}^d)$, and assume that p_i and p_j in P verify " $p_i - p_j$ "₂ $\leq r$. Let C be the connected component of $G_{\text{disk}}(r)$ to which they belong. With a slight abuse of notation, we also denote by C the vertices of the connected component. Since C is connected, then $G_{\text{EMST}}(C) \subset C$ by fact (i). Moreover, since all the nodes in C are at a distance strictly larger than r from any node of C , we deduce from the definition of the Euclidean minimum spanning tree that $G_{\text{EMST}}(C)$ is equal to the subgraph of $G_{\text{EMST}}(P)$ induced by C . Therefore, $G_{\text{EMST}}(C) \subset G_{\text{EMST}} \cap G_{\text{disk}}(r)(P)$, and p_i and

p_j belong to the same component of $\mathcal{G}_{\text{EMST} \cap \text{disk}(r)}(\mathcal{P})$. This implies the result. $\square$

2.5.2 Proof of Proposition 2.9

Proof. Regarding the statement on $\mathcal{G}_{\text{RN}} \cap \mathcal{G}_{\text{disk}}(r)$, note that

$$B(p_i, \|p_i - p_j\|_2) \cap B(p_j, \|p_i - p_j\|_2) \subset B(p_i, \|p_i - p_j\|_2).$$

Therefore, if $\|p_i - p_j\|_2 \leq r$, then any node contained in the intersection $B(p_i, \|p_i - p_j\|_2) \cap B(p_j, \|p_i - p_j\|_2)$ must necessarily be within a distance r of p_i . From here, we deduce that $\mathcal{G}_{\text{RN}} \cap \mathcal{G}_{\text{disk}}(r)$ is spatially distributed over $\mathcal{G}_{\text{disk}}(r)$. Regarding the statement on $\mathcal{G}_{\text{G}} \cap \mathcal{G}_{\text{disk}}(r)$, note that

$$B\left(\frac{p_i + p_j}{2}, \frac{\|p_i - p_j\|_2}{2}\right) \subset B(p_i, \|p_i - p_j\|_2).$$

Therefore, if $\|p_i - p_j\|_2 \leq r$, then any node contained in $B\left(\frac{p_i + p_j}{2}, \frac{\|p_i - p_j\|_2}{2}\right)$, must necessarily be within a distance r of p_i . From here, we deduce that $\mathcal{G}_{\text{G}} \cap \mathcal{G}_{\text{disk}}(r)$ is spatially distributed over $\mathcal{G}_{\text{disk}}(r)$. Finally, note that if $\|p_i - p_j\|_2 > r$, then the half-plane $\{q \in \mathbb{R}^2 \mid \|q - p_i\|_2 \leq \|q - p_j\|_2\}$ contains the ball $B(p_i, \frac{r}{2})$. Accordingly,

$$\begin{aligned} V_{i, \frac{r}{2}}(\mathcal{P}) &= V_i(\mathcal{P}) \cap \overline{B(p_i, \frac{r}{2})} \\ &= \{q \in \mathbb{R}^2 \mid \|q - p_i\|_2 \leq \|q - p_j\|_2, \text{ for all } p_j \in \mathcal{P} \cap \overline{B(p_i, \frac{r}{2})}\} \\ &= \{q \in \mathbb{R}^2 \mid \|q - p_i\|_2 \leq \|q - p_j\|_2, \text{ for all } p_j \in \mathcal{N}_{\mathcal{G}_{\text{disk}}(r), p}(\mathcal{P})\} \cap \overline{B(p_i, \frac{r}{2})} \end{aligned}$$

from which we deduce that $\mathcal{G}_{\text{LD}}(r)$ is spatially distributed over $\mathcal{G}_{\text{disk}}(r)$. $\square$

2.5.3 Proof of Theorem 2.16

We begin with some preliminary notions. In the following, a set $\Omega \subset \mathbb{R}^2$ is *piecewise continuously differentiable* if its boundary, $\partial\Omega$, is a not self-intersecting closed curve that admits a continuous and piecewise continuously differentiable parameterization $\gamma : [0, 1] \rightarrow \mathbb{R}^2$. Likewise, a collection of sets $\{\Omega(x) \subset \mathbb{R}^2 \mid x \in (a, b)\}$ is a *piecewise continuously differentiable family* if $\Omega(x)$ is piecewise continuously differentiable for all $x \in (a, b)$, and there exists a continuous function $\gamma : [0, 1] \times (a, b) \rightarrow \mathbb{R}^2$, $(t, x) \mapsto \gamma(t, x)$, continuously differentiable with respect to its second argument, such that for each $x \in (a, b)$, the map $t \mapsto \gamma_x(t) = \gamma(t, x)$ is a continuous and piecewise continuously differentiable parameterization of $\partial\Omega(x)$. We refer to γ as a *parameterization for the family* $\{\Omega(x) \subset \mathbb{R}^2 \mid x \in (a, b)\}$.

The following result is an extension of the Law of Conservation of Mass in fluid mechanics (Chorin and Marsden, 1994) and of the classic divergence theorem in differential geometry (Chavel, 1984).

Proposition 2.23 (Generalized conservation of mass). *Let $\{\Omega(x) \subset \mathbb{R}^2 \mid x \in (a, b)\}$ be a family of star-shaped sets with piecewise continuously differentiable boundary. Let the function $\varphi : \mathbb{R}^2 \times (a, b) \rightarrow \mathbb{R}$ be continuous on $\mathbb{R}^2 \times (a, b)$ that is continuously differentiable with respect to its second argument for all $x \in (a, b)$ and almost all $q \in \Omega(x)$, and such that for each $x \in (a, b)$, the maps $q \mapsto \varphi(q, x)$ and $q \mapsto \frac{\partial \varphi}{\partial x}(q, x)$ are measurable, and integrable on $\Omega(x)$. Then, the function*

$$(a, b) \ni x \mapsto \int_{\Omega(x)} \varphi(q, x) dq \quad (2.5.1)$$

is continuously differentiable and

$$\frac{d}{dx} \int_{\Omega(x)} \varphi(q, x) dq = \int_{\Omega(x)} \frac{\partial \varphi}{\partial x}(q, x) dq + \int_{\partial \Omega(x)} \varphi(\gamma, x) n(\gamma) \cdot \frac{\partial \gamma}{\partial x} d\gamma,$$

where $n : \partial \Omega(x) \rightarrow \mathbb{R}^2$, $q \mapsto n(q)$, denotes the unit outward normal to $\partial \Omega(x)$ at $q \in \partial \Omega(x)$, and $\gamma : [0, 1] \times (a, b) \rightarrow \mathbb{R}^2$ is a parameterization for the family $\{\Omega(x) \subset \mathbb{R}^2 \mid x \in (a, b)\}$.

We interpret the proposition as follows: in the fluid mechanics interpretation, as the parameter x changes, the total mass variation inside the region can be decomposed into two terms. The first term is the amount of mass created inside the region, whereas the second term is the amount of mass that crosses the moving boundary of the region.

Proof of Proposition 2.23. Let $x_0 \in (a, b)$. Using the fact that the map γ is continuous and that $\Omega(x_0)$ is star-shaped, one can show that there exist an interval around

x_0 of the form $(x_0 - \varepsilon, x_0 + \varepsilon)$, a continuously differentiable function $u_{x_0} : [0, 1] \times \mathbf{R}_{\geq 0} \rightarrow \mathbf{R}$ and a function $r_x : [0, 1] \times (x_0 - \varepsilon, x_0 + \varepsilon) \rightarrow \mathbf{R}_{\geq 0}$ continuously differentiable in its second argument and piecewise continuously differentiable in its first argument, such that for all $x \in (x_0 - \varepsilon, x_0 + \varepsilon)$, one has

$$\Omega(x) = \cup_{t \in [0, 1]} \{u_{x_0}(t, s) \mid 0 \leq s \leq r_{x_0}(t, x)\},$$

$$\gamma(t, x) = u_{x_0}(t, r_{x_0}(t, x)), \quad \text{for all } t \in [0, 1].$$

For simplicity, we denote by r and u the functions r_{x_0} and u_{x_0} , respectively. By definition, the function in (2.5.1) is continuously differentiable at x_0 if the following limit exists:

$$\lim_{h \rightarrow 0} \frac{1}{h} \cdot \int_{\Omega(x_0+h)} \varphi(q, x_0+h) dq - \int_{\Omega(x_0)} \varphi(q, x_0) dq = \Sigma,$$

and depends continuously on x_0 . Now, we can rewrite the previous limit as

$$\begin{aligned} & \lim_{h \rightarrow 0} \frac{1}{h} \int_0^1 \int_0^{r(t, x_0+h)} \varphi(u(t, s), x_0+h) \frac{\partial u}{\partial t} \times \frac{\partial u}{\partial s} ds \\ & - \frac{1}{h} \int_0^1 \int_0^{r(t, x_0)} \varphi(u(t, s), x_0) \frac{\partial u}{\partial t} \times \frac{\partial u}{\partial s} ds dt \\ & + \lim_{h \rightarrow 0} \frac{1}{h} \int_0^1 \int_0^{r(t, x_0+h)} \varphi(u(t, s), x_0+h) \frac{\partial u}{\partial t} \times \frac{\partial u}{\partial s} ds \\ & - \int_0^1 \int_0^{r(t, x_0)} \varphi(u(t, s), x_0) \frac{\partial u}{\partial t} \times \frac{\partial u}{\partial s} ds dt, \quad (2.5.2) \end{aligned}$$

where $\times$ denotes the vector product and for brevity we omit the fact that

the partial derivatives $\frac{\partial u}{\partial t}$ and $\frac{\partial u}{\partial s}$ are evaluated at (t, s) in the integrals.

Regarding the second integral in the last equality of (2.5.2), since

$$\begin{aligned} & \lim_{h \rightarrow 0} \frac{1}{h} \int_0^1 \int_0^{r(t, x_0+h)} (\varphi(u(t, s), x_0+h) - \varphi(u(t, s), x_0)) \frac{\partial u}{\partial t} \times \frac{\partial u}{\partial s} ds \\ & = \frac{\partial \varphi}{\partial x_0}(u(t, s), x_0) \frac{\partial u}{\partial t} \times \frac{\partial u}{\partial s}, \end{aligned}$$

almost everywhere, and this function is measurable and its integral over the bounded set $\Omega(x_0)$ is finite by hypothesis, the Lebesgue Dominated Convergence Theorem (Bartle, 1995) implies that

$$\liminf \int_0^1 \int_0^{r(t, x_0)} (\varphi(u(t, s), x_0+h) - \varphi(u(t, s), x_0)) \frac{\partial u}{\partial t} \times \frac{\partial u}{\partial s} ds dt$$

$$\begin{aligned}
& \lim_{h \rightarrow 0} \int_0^1 \int_{x_0}^{r(t, x_0+h)} \frac{\partial u}{\partial t}(t, s) \times \frac{\partial u}{\partial s}(t, s) ds dt \\
&= \int_0^1 \int_{x_0}^{r(t, x_0)} \frac{\partial \varphi}{\partial x}(u(t, s), x_0) \frac{\partial u}{\partial t}(t, s) \times \frac{\partial u}{\partial s}(t, s) ds dt \\
&= \int_{\Omega(x_0)} \frac{\partial \varphi}{\partial x}(q, x_0) dq. \tag{2.5.3}
\end{aligned}$$

On the other hand, regarding the first integral in the last equality of (2.5.2), using the continuity of φ , one can deduce that

$$\begin{aligned}
& \lim_{h \rightarrow 0} \int_0^1 \int_{x_0}^{r(t, x_0)} \frac{\partial u}{\partial t}(t, s) \times \frac{\partial u}{\partial s}(t, s) ds dt \\
&= \lim_{h \rightarrow 0} \int_0^1 \int_{x_0}^{r(t, x_0)} \frac{\partial u}{\partial t}(t, r(t, z)) \times \frac{\partial u}{\partial s}(t, r(t, z)) \frac{\partial r}{\partial z}(t, z) dz dt \\
&= \int_0^1 \frac{\partial u}{\partial t}(t, r(t, x_0)) \times \frac{\partial u}{\partial s}(t, r(t, x_0)) \frac{\partial r}{\partial x_0}(t, x_0) dt.
\end{aligned}$$

Since $\gamma(t, x) = u(t, r(t, x))$ for all $t \in [0, 1]$ and $x \in (x_0 - \varepsilon, x_0 + \varepsilon)$, one has

$$\begin{aligned} \frac{\partial \gamma}{\partial t}(t, x_0) &= \frac{\partial u}{\partial t}(t, r(t, x_0)) + \frac{\partial u}{\partial s}(t, r(t, x_0)) \frac{\partial r}{\partial t}(t, x_0), \\ \frac{\partial \gamma}{\partial x}(t, x_0) &= \frac{\partial u}{\partial s}(t, r(t, x_0)) \frac{\partial r}{\partial x}(t, x_0). \end{aligned}$$

Let χ denote the angle formed by $\frac{\partial \gamma}{\partial t}(t, x_0)$ and $\frac{\partial u}{\partial s}(t, r(t, x_0))$. Then (omit-

ting the expression $(t, r(t, x))$ for brevity),

$$\begin{aligned} \frac{\partial u}{\partial t} \times_{\partial s} \frac{\partial u}{\partial s} &= \left(\frac{\partial u}{\partial t} + \frac{\partial u}{\partial s} \frac{\partial r}{\partial t} \right) \times \frac{\partial u}{\partial s} \\ &= \frac{\partial \gamma}{\partial t} \frac{\partial u}{\partial s} \sin \chi = \frac{\partial \gamma}{\partial t} n^T(\gamma) \frac{\partial u}{\partial s}, \end{aligned}$$

where in the last inequality we have used the fact that, since γ_{x_0} is a parameterization of $\partial\Omega(x_0)$, then $\sin \chi = \cos \psi$, where ψ is the angle formed by n , the outward normal to $\partial\Omega(x_0)$, and $\frac{\partial u}{\partial s}$. Therefore, we finally arrive

$$\begin{aligned} &\int_0^1 \varphi(\gamma(t), x_0) \frac{\partial u}{\partial t}(t, r(t, x_0)) \times \frac{\partial u}{\partial s}(t, r(t, x_0)) \frac{\partial r}{\partial x}(t, x_0) dt \\ &= \int_0^1 \varphi(\gamma(t), x) \frac{\partial \gamma}{\partial t}(t, x) n^T(\gamma(t, x)) \frac{\partial \gamma}{\partial x}(t, x_0) dt \\ &= \int_{\partial\Omega(x_0)} \varphi(\gamma, x_0) n^T(\gamma) \frac{\partial \gamma}{\partial x} d\gamma. \end{aligned} \quad (2.5.4)$$

Given the hypothesis of Proposition 2.23, both terms in (2.5.3) and (2.5.4) have a continuous dependence on $x_0 \in (a, b)$. This concludes the proof. $\square$

We are finally ready to state the proof of the main result of Section 2.3.

Proof of Theorem 2.16. We prove the theorem statement

when the performance function is continuously differentiable and we refer to Cortés et al. (2005) for the complete proof for the case when the performance function

is piecewise continuously differentiable. Specifically, we show that if f is continuously differentiable, then for $P \in$

$$S^n \setminus S_{\text{coinc}}, \quad \frac{\partial H_{\text{exp}}}{\partial p_i}(P) = \int_{V_i(P)} \frac{\partial}{\partial p_i} f(\|q - p\|^2) \varphi(q) dq.$$

From Proposition 2.23, we have

$$\frac{\partial}{\partial p} \sum_{i=1}^n \int_{V_i(P)} f(\|q - p_i\|) \varphi(q) dq = \sum_{i=1}^n \int_{V_i(P)} \frac{\partial}{\partial p} f(\|q - p_i\|) \varphi(q) dq + \sum_{j=1}^n \int_{\partial V_j(P)} \phi(p_j, q) \cdot n(\gamma_j) \cdot \frac{\partial \gamma_j}{\partial p} d\gamma_j,$$

where γ_j is a parametrization of $V_j(P)$ and where we abbreviate $\phi(p_j, q) = f(\|q - p_j\|) \varphi(q)$. Next, we show that the second term vanishes. Note that the motion of p_i affects the Voronoi cell $V_i(P)$ and the cells of all its neighbors in $N_{G_D, p_i}(P)$. Therefore, the second term equals

$$\int_{\partial V_i(P)} \phi(p_i, q) \cdot n(\gamma_i) \cdot \frac{\partial \gamma_i}{\partial p} d\gamma_i + \sum_{p_j \in N_{G_D, p_i}(P)} \int_{\partial V_j(P)} \phi(p_j, q) \cdot n(\gamma_j) \cdot \frac{\partial \gamma_j}{\partial p} d\gamma_j.$$

Without loss of generality, assume that $V_i(P)$ does not share any face with

∂S . Since the boundary of $V_i(P)$ satisfies $\partial V_i(P) = \sum_j \Delta_{ij}$, where $\Delta_{ij} = \Delta_{ji}$

is the edge between $V_i(P)$ and $V_j(P)$, for all neighbors p_j , we compute

$$\int_{\partial V_i(P)} \phi(p_i, q) \cdot n(\gamma_i) \cdot \frac{\partial \gamma_i}{\partial p} d\gamma_i = \sum_{p_j \in N_{G_D, p_i}(P)} \int_{\Delta_{ij}} \phi(p_i, q) \cdot n_{ij}(\gamma_j) \cdot \frac{\partial \gamma_j}{\partial p} d\gamma_j + \sum_{p_j \in N_{G_D, p_i}(P)} \int_{\Delta_{ji}} \phi(p_j, q) \cdot n_{ji}(\gamma_j) \cdot \frac{\partial \gamma_j}{\partial p} d\gamma_j,$$

where n_{ij} denotes the unit normal along Δ_{ij} outward of $V_i(P)$.

Noting that

$n_{ji} = -n_{ij}$ and collecting the results obtained so far, we write

$$\sum_{j=1}^n \phi(p_j, q) n(\gamma_j) \cdot \frac{\partial \gamma_j}{\partial p} = \sum_{p_i \in N_G(p)} \int_{\Delta_{ij}} \phi(p_i, q) - \phi(p_j, q) n_{ij}(\gamma_j) \cdot \frac{\partial \gamma_j}{\partial p} d\gamma_j.$$

This quantity vanishes because $f(q - p_i) = f(q - p_j)$, and therefore

$\phi(p_i, q) = \phi(p_j, q)$ for any q belonging to the edge Δ_{ij} . □

2.5.4 Proof of Proposition 2.19

Proof. Recall that $H_{dc}(p_1, \dots, p_n) = H_{dc}(p_1, \dots, p_n, V_1(P), \dots, V_n(P))$. To show the first inequality, let $j \in \{1, \dots, n\}$ and $q_* \in V_j(P)$ be such that

$H_{dc}(p_1, \dots, p_n) = \|q_* - p_j\|^2$. By definition, given a partition $\{W_1, \dots, W_n\}$

of S , there exists k such that $q_* \in W_k$. Therefore,

$$\begin{aligned} H_{dc}(p_1, \dots, p_n) &= \|q_* - p_j\|^2 \leq \|q_* - p_k\|^2 \\ &\leq \max_{q \in W_k} \|q - p_j\|^2 \leq H_{dc}(p_1, \dots, p_n, W_1, \dots, W_n). \end{aligned}$$

To show the second inequality, note that the definition of circumcenter implies that, for each $i \in \{1, \dots, n\}$,

$$\max_{q \in \partial W_i} \|q - CC(W_i)\|^2 \leq \max_{q \in \partial W_i} \|q - p_i\|^2.$$

Taking the maximum over all nodes, we deduce that

$$H_{dc}(CC(W_1), \dots, CC(W_n), W_1, \dots, W_n) \leq H_{dc}(p_1, \dots, p_n, W_1, \dots, W_n),$$

as claimed. $\square$

2.5.5 Proof of Proposition 2.21

Proof. Recall that $H_{sp}(p_1, \dots, p_n) = H_{sp}(p_1, \dots, p_n, V_1(P), \dots, V_n(P))$. To show the first inequality, let $j \in \{1, \dots, n\}$ and $q_* \in \text{int}(V_j(P))$ be such that $H_{sp}(p_1, \dots, p_n) = \|q_* - p_j\|^2$. Since $q_* \in \text{int}(V_j(P))$, there exists $i \in \{1, \dots, n\}$ such that $\|q_* - p_j\|^2 \geq \|q_* - p_i\|^2$. On the other hand, there must exist $k \in \{1, \dots, n\}$ such that $q_* \in W_k$. Now if $k = j$, then $q_* \in \text{int}(W_j)$. Therefore,

$$\begin{aligned} H_{sp}(p_1, \dots, p_n) &= \|q_* - p_j\|^2 \geq \|q_* - p_i\|^2 \\ &\geq \min_{q \in \text{int}(W_i)} \|q - p_i\|^2 \geq H_{sp}(p_1, \dots, p_n, W_1, \dots, W_n). \end{aligned}$$

Now, if $k = i$, then $q_* \in \text{int}(W_j)$. Therefore,

$$H_{sp}(P) = \|q_* - p_j\|^2 \geq \min_{q \in \text{int}(W_j)} \|q - p_i\|^2 \geq H_{sp}(p_1, \dots, p_n, W_1, \dots, W_n).$$

Finally, if $k \neq i, j$, then $q_* \in \text{int}(W_i) \cup \text{int}(W_j)$, and a similar argument guarantees $H_{sp}(p_1, \dots, p_n) \geq H_{sp}(p_1, \dots, p_n, W_1, \dots, W_n)$.

To show the second inequality, let $i \in \{1, \dots, n\}$ and select $q_i \in \text{IC}(W_i)$.

The definition of the incenter set implies that,

$$\min_{q \in \partial W_i} \|q - q_i\|^2 \geq \min_q \|q - p_i\|^2.$$

The expression on the left does not depend on the specific point selected in the incenter set. Taking the minimum over all nodes, we deduce that

$$H_{\text{sp}}(\text{IC}(W_1), \dots, \text{IC}(W_n), W_1, \dots, W_n) \geq H_{\text{sp}}(p_1, \dots, p_n, W_1, \dots, W_n),$$

as claimed. $\square$

6.Related Work

Chapter Three

Robotic network models and complexity notions

This chapter introduces the main subject of study of this book, namely a model for groups of robots that sense their own position, exchange messages according to a geometric communication topology, process information, and control their motion. We refer to such systems as robotic networks. The content of this chapter has evolved from [Martínez et al. \(2007a\)](#).

The chapter is organized as follows. The first section contains the formal model. We begin by presenting the physical components of a network, that is, the mobile robots and the communication service connecting them. We then present the notion of control and communication law, and how a law is executed by a robotic network. These notions subsume the notions of synchronous network and distributed algorithm described in Section 1.5. As an example of these notions, we introduce a simple law, called the agree and pursue law, which combines ideas from leader election algorithms and from cyclic pursuit (i.e., a game in which

robots chase each other in a circular environment). In the second section, we propose a model of groups of robots that interact through sensing, rather than communication. The third section discusses time, space, and communication complexity notions for robotic networks as extensions of the corresponding notions for distributed algorithms. The complexity notions rely on the basic concept of coordination task and task achievement. The fourth and last section establishes the time, space, and communication required by the agree and pursue law to steer a group of robots to a uniformly spaced rotating configuration. We end the chapter with three sections on, respectively, bibliographical notes, proofs of the results presented in the chapter, and exercises.

3.1 A MODEL FOR SYNCHRONOUS ROBOTIC NETWORKS

Here, we introduce a model for a synchronous robotic network. This model is an extension of the synchronous network model presented in Section 1.5.1. We start by detailing the physical components of the network, which include the robots themselves as well as the communication service among them.

3.1.1 Physical components

We start by providing a basic definition of a robot and a model for how each robot moves in space.

A *mobile robot* is a continuous-time continuous-space dynamical system as defined in Section 1.3, that is, a tuple (X, U, X_0, f) , where

- (i) X is d -dimensional space chosen among $\mathbb{R}^d, S^d$, and the Cartesian products $\mathbb{R}^{d_1} \times S^{d_2}$, for some $d_1 + d_2 = d$, called the *state space*;
- (ii) U is a compact subset of $\mathbb{R}^m$ containing 0_m , called the *input space*;
- (iii) X_0 is a subset of X , called the *set of allowable initial states*; and
- (iv) $f: X \times U \rightarrow \mathbb{R}^d$ is a continuously differentiable control vector field on X , that is, f determines the robot motion $x: \mathbb{R}_{\geq 0} \rightarrow X$ via the differential equation, or control system, $\rightarrow$

$$\dot{x}(t) = f(x(t), u(t)), \quad (3.1.1)$$

subject to the control $u: \mathbb{R}_{\geq 0} \rightarrow U$.

We will use the terms “robot” and “agent” interchangeably. We refer to $x \in X$ and $u \in U$ as a *physical state* and an *input* of the mobile robot, respectively. Most often, the physical state will have the interpretation of a location, or a location and velocity. We will often consider control-affine vector fields. In such a case, we represent f as the ordered family of continuously differentiable vector fields $(f_0, f_1, \dots, f_m)$ on X . In general, the control signal u will not depend only on time but also on x and possibly other variables in the system. Note that there is no additional difficulty in modeling mobile robots using dynamical systems defined on manifolds (Bullo and Lewis, 2004), but we avoid it here in the interest of simplicity.

Example 3.1 (Planar vehicle models). The following models of control systems are commonly used in robotics, beginning with the early works of Dubins (1957), and Reeds and Shepp (1990). Figures 3.1(a) and

(b) show a two-wheeled vehicle and a four-wheeled vehicle, respectively. The two-wheeled planar vehicle is described by the dynamical system

$$\dot{x} = v \cos \theta, \quad \dot{y} = v \sin \theta, \quad \dot{\theta} = \omega, \quad (3.1.2)$$

with state variables $x \in \mathbf{R}$, $y \in \mathbf{R}$, and $\theta \in S^1$, describing the planar position and orientation of the vehicle, and with controls v and ω , describing the forward linear velocity and the angular velocity of the vehicle. Depending on which set the controls are restricted to, we define the following models:

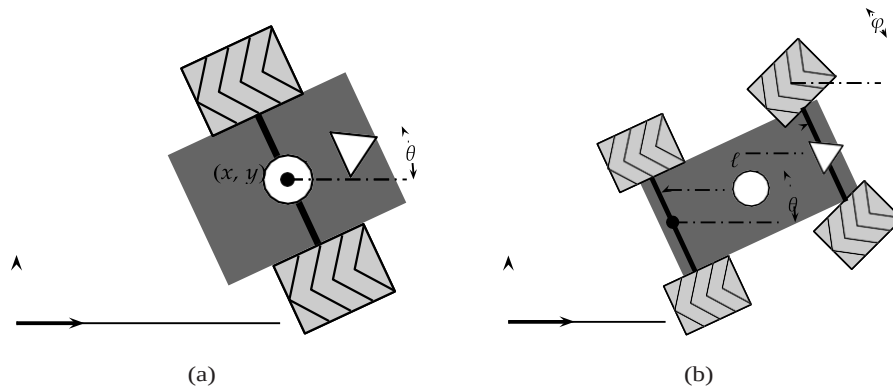


Figure 3.1 A two-wheeled vehicle (a) and four-wheeled vehicle (b). In each case, the orientation of the vehicle is indicated by the small triangle.

The unicycle. The controls v and ω take value in $[-1, 1]$ and $[-1, 1]$, respectively.

The differential drive robot. Set $v = (\omega_{\text{right}} + \omega_{\text{left}})/2$ and $\omega = (\omega_{\text{right}} - \omega_{\text{left}})/2$ and assume that both ω_{right} and ω_{left} take value in $[-1, 1]$.

The Reeds - Shepp car. The control v takes values in $\{-1, 0, 1\}$ and the control ω takes values in $[-1, 1]$.

The Dubins vehicle. The control v is set equal to 1 and the control ω takes value in $[-1, 1]$.

Finally, the four-wheeled planar vehicle, composed of a front and a rear axle separated by a distance ℓ , is described by the same dynamical system (3.1.2) with the following distinctions: $(x, y) \in \mathbb{R}^2$ is the position of the midpoint of the rear axle, $\theta \in S^1$ is the orientation of the rear axle, the control v is the forward linear velocity of the rear axle, and the angular velocity satisfies $\omega = \frac{v}{\ell} \tan \phi$, where the control ϕ is the steering angle of the vehicle.

Next, we generalize the notion of synchronous network introduced in Definition 1.38 and introduce a corresponding notion of robotic network.

Definition 3.2 (Robotic network). The physical components of a *robotic network* $\mathbf{S}$ consist of a tuple $(I, \mathbf{R}, E_{\text{cmm}})$, where

- (i) $I = \{1, \dots, n\}$, I is called the *set of unique identifiers (UIDs)*;
- (ii) $\mathbf{R} = \{R^{[i]}\}_{i \in I} = \{(X^{[i]}, U^{[i]}, X_{[i]}^{[i]})\}_{i \in I}$ is a set of mobile robots;
- (iii) E_{cmm} is a map from $\mathcal{Q}_{I \times I}$ to the subsets of $I \times I$ —this map is called the *communication edge map*.

Additionally, if all mobile robots are identical, that is, if $R^{[i]} = (X, U, X_0, f)$ for all $i \in \{1, \dots, n\}$, then the robotic network is *uniform*.

Remarks 3.3 (Notational conventions and meaning of the communication edge map).

- (i) Following the convention established in Section 1.5, we let the superscript $[i]$ denote the variables and spaces which correspond to the robot with unique identifier i ; for instance, $x_0^{[i]} \in X^{[i]}$ and $x^{[i]} \in X^{[i]}$ denote the physical state and the initial physical state of robot $R^{[i]}$, respectively. We refer to $(x^{[1]}, \dots, x^{[n]}) \in \prod_{i \in I} X^{[i]}$ as a *state* $x = (x^{[1]}, \dots, x^{[n]})$ of the network.
- (ii) The map $x \mapsto (I, E_{\text{cmm}}(x))$ models the topology of the communication service among the robots: at a physical state $x = (x^{[1]}, \dots, x^{[n]})$, two robots at locations $x^{[i]}$ and $x^{[j]}$ can communicate if and only if the pair (i, j) is an edge in $E_{\text{cmm}}(x) = E_{\text{cmm}}(x^{[1]}, \dots, x^{[n]})$. Accordingly, we refer to $(I, E_{\text{cmm}}(x))$ as the *communication graph* at x .

When and which robots communicate is discussed in Section 3.1.2. As communication graphs, we will often adopt one of the proximity graphs discussed in Section 2.2, and in particular the (undirected) disk graph.

To make things concrete, let us present some examples of robotic networks that will be commonly used later.

Example 3.4 (First-order robots with range-limited communication). Consider a group of robots moving in $\mathbb{R}^d$, $d \geq 1$. As in Chapter 2, we let p denote a point in $\mathbb{R}^d$ and we let $p^{[1]}, \dots, p^{[n]}$ denote the robot locations. Assume that the robots move according to

$$\dot{p}^{[i]}(t) = u^{[i]}(t), \quad (3.1.3)$$

with $u^{[i]} \in [-u_{\max}, u_{\max}]^d$; for an illustration, see Figure 3.2. According to our mobile robot notation, these are identical robots of the form

$$(\mathbb{R}^d, [-u_{\max}, u_{\max}]^d, \mathbb{R}^d, (0_d, e_1, \dots, e_d)).$$

We assume that each robot can sense its own position and can communicate with any other robot within distance r , that is, we adopt the r -disk graph

$G_{\text{disk}}(r)$ defined in Section 2.2 as communication graph. These data define the uniform robotic network S_{disk} .

It will also be interesting to consider first-order robots with communication graphs other than the disk graph; important examples include the Delaunay graph G_D , the limited Delaunay graph $G_{LD}(r)$, and the ∞ -disk

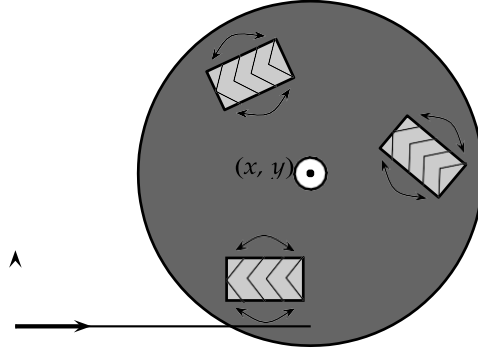


Figure 3.2 An omnidirectional vehicle. In addition to controlling the rotation speed of the wheels, the vehicle can also actuate the direction in which they point. This allows the vehicle to move in any direction according to the first-order dynamics (3.1.3).

graph $G_{\infty\text{-disk}}(r)$, discussed in Section 2.2. These three graphs, adopted as communication models, give rise to three robotic networks denoted S_D , and S_{LD} , $S_{\infty\text{-disk}}$, respectively.

•

Example 3.5 (Planar vehicle robots with Delaunay communication). We consider a group of vehicle robots moving in an allowable environment $Q \subset \mathbb{R}^2$ according to the planar vehicle dynamics introduced in Example 3.1. We let $\{(p^{[1]}, \theta^{[1]}), \dots, (p^{[n]}, \theta^{[n]})\}$ denote the robot physical states, where $p^{[i]} = (x^{[i]}, y^{[i]}) \in Q$ corresponds to the position and $\theta^{[i]} \in S^1$ corresponds to the orientation of the robot $i \in I$. As the communication graph, we adopt the Delaunay graph G_D on Q introduced in Section 2.2. These data define the uniform robotic network S_{vehicles} .

•

Example 3.6 (Robots with line-of-sight communication). We consider a group of robots moving in an allowable environment $Q \subset \mathbb{R}^2$. As in Example 3.4, we let $p^{[1]}, \dots, p^{[n]}$ denote the robot

locations and we assume that the robots move according to the motion model (3.1.3). Each robot can sense its own position and the boundary of ∂Q , and can communicate with any other robot within distance r and within line of sight, that is, we adopt the range-limited visibility graph $G_{\text{vis-disk},Q}$ in Q defined in Section 2.2 as the communication graph. These data define the uniform robotic network $S_{\text{vis-disk}}$.

•
 Example 3.7 (First-order robots in S^1). Consider a group of n robots $\theta^{[1]}, \dots, \theta^{[n]}$ in S^1 , moving along on the unit circle with an angular velocity equal to the control input. Each identical robot is described by the tuple $(S^1, [u_{\max}, u_{\max}], S^1, (0, e))$, where e is the vector field on S^1 describing unit-speed counterclockwise rotation. As in the previous examples, we

assume that each robot can sense its own position and can communicate with any other robot within distance r along the circle, that is, we adopt the r -disk graph $G_{\text{disk}}(r)$ on S^1 defined in Section 2.2 as the communication graph. These data define the uniform robotic network S_{circle} .

.

We conclude this section with a remark.

Remark 3.8 (Congestion models in robotic networks). The behavior of a robotic network might be affected by communication and physical congestion problems.

Communication congestion: Omnidirectional wireless transmissions interfere. Clear reception of a signal requires that no other signals are present at the same point in time and space. In an *ad hoc* network, node i receives a message transmitted by node j only if all other neighbors of i are silent. In other words, the transmission medium is shared among the agents. As the density of agents increases, so does wireless communication congestion. The following asymptotic and optimization results are known.

First, for *ad hoc* networks with n uniformly randomly placed nodes, it is known (Gupta and Kumar, 2000) that the maximum-throughput communication range $r(n)$ of each node decreases as the density of nodes increases; in d dimensions, the appropriate scaling law is $r(n)$

$\Theta((\log(n)/n)^{1/d})$. This is referred to as the *connectivity regime* in percolation theory and statistical mechanics. Using the k -nearest neighbor graph over uniformly placed nodes, the analysis in Xue and Kumar (2004) suggests that the minimal number of neighbors in a connected network grows with $\log(n)$.

Second, a growing body of literature (Santi, 2005;

Lloyd et al., 2005) is available on *topology control*, that is, on how to compute transmission power values in an *ad hoc* network so as to minimize energy consumption and interference (due to multiple sources), while achieving various graph topological properties, such as connectivity or low network diameter.

Physical congestion. Robots can collide: it is clearly important to avoid “simultaneous access to the same physical area” by multiple robots. It is reasonable to assume that, as the number of robots increases, so should the area available for their motion. A convenient alternative approach is the one taken by Sharma et al. (2007), where robots’ safety zones decrease with decreasing robot speed. This suggests that, in a fixed environment, individual nodes of a large ensemble have to move at a speed decreasing with n , and in particular, at a speed proportional to $n^{-1/d}$. Roughly speaking, if the overall volume V in which the

groups of agents move is constant, and there are n robots, then the speed v at which they can move goes approximately as $v^d \approx \frac{v}{n}$.

In summary, one way to incorporate congestion effects into the robotic network model is to assume that the parameters of the network physical components depend upon the number of robots n . In the limit as $n \rightarrow +\infty$, we will sometimes assume that r and $u_{\max}$, the communication range and the velocity upper bound in Examples 3.4 and 3.7, are of order $n^{-1/d}$.

3.1.2 Control and communication laws

Here, we present a discrete-time communication, continuous-time motion model for the evolution of a robotic network subject to a communication and control law. In our model, each robot evolves in the physical domain in continuous time, senses its position in continuous time, and, in discrete time, exchanges information with other robots and executes a state machine, which we shall refer to as a processor. The following definition is a generalization of the concept of distributed algorithm introduced in Definition 1.39 and of the classical notion of dynamical feedback controller.

Definition 3.9 (Control and communication law). A *control and communication law* CC for a robotic network S consists of the sets:

- (i) A , a set containing the null element, called the *communication alphabet*—elements of A are called *messages*;
- (ii) $W^{[i]}, i \in I$, called the *processor state sets*; and
- (iii) $W_0^{[i]} \subseteq W^{[i]}, i \in I$, sets of *allowable initial values*;

and of the following maps:

- (i) $\text{msg}^{[i]} : X^{[i]} \times W^{[i]} \times \prod_{j \in I} W^{[j]} \rightarrow A, i \in I$, called *message-generation functions*;
- (ii) $\text{stf}^{[i]} : X^{[i]} \times W^{[i]} \times \prod_{j \in I} A^j \rightarrow W^{[i]}, i \in I$, called *(processor) state-*

(iii) $\text{ctl}^{[i]} : X^{[i]} \times \prod_{\substack{X^{[i]} \\ \subset}} W^{[i]} \times A^n \rightarrow U^{[i]}, i \in I$, called *(motion) control functions*; and

If S is uniform and if $W^{[i]} = W$, $\text{msg}^{[i]} = \text{msg}$, $\text{stf}^{[i]} = \text{stf}$, and $\text{ctl}_0^{[i]} = \text{ctl}$, for all $i \in I$, then CC is said to be *uniform* and is described by a tuple $(A, W, \{W^{[i]}\}_{i \in I}, \text{msg}, \text{stf}, \text{ctl})$.

•

We will sometimes refer to a control and communication law as a *distributed motion coordination algorithm*. Roughly speaking, the rationale behind Definition 3.9 is as follows (see Figure 3.3). The state of robot i in-

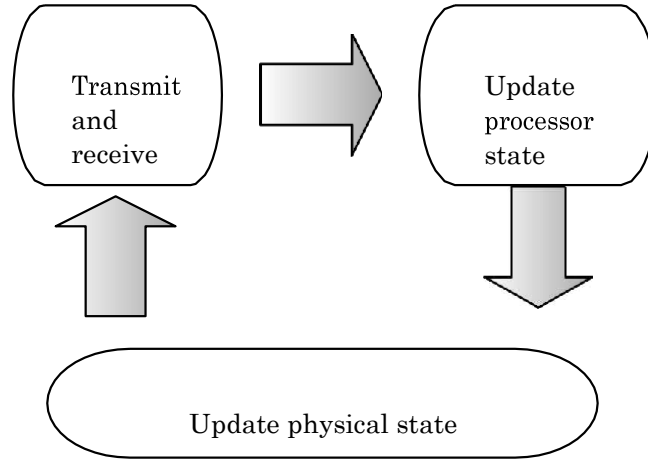


Figure 3.3 The execution of a control and communication law by a robotic network.

cludes both the physical state $x^{[i]} \in X^{[i]}$ and the *processor state* $w^{[i]} \in W^{[i]}$ of the state machine that robot i implements. These states are initialized with values in their corresponding allowable initial sets $X^{[i]}$ and $W^{[i]}$. We

assume that the robot can sense its own physical position $x^{[i]}$. At each time instant $\ell \in \mathbb{Z}_{\geq 0}$, robot i sends to each of its out-neighbors j in the communication digraph $(I, E_{\text{cmp}}(x))$ a message (possibly the null message) computed by applying the message-generation function $\text{msg}^{[i]}$ to the current values of its physical state $x^{[i]}$ and processor state $w^{[i]}$, and to the identity j . Subsequently, but still at the time instant $\ell \in \mathbb{Z}_{\geq 0}$, robot i updates the value of its processor state $w^{[i]}$ by applying the state-transition function $\text{stf}^{[i]}$ to the current value of its physical state $x^{[i]}$, processor state $w^{[i]}$ and to the messages it receives from its in-neighbors. Between communication instants, that is, for $\ell \in [\ell, \ell + 1)$ for some $\ell \in \mathbb{Z}_{\geq 0}$, the motion of the i th robot is determined by applying the control function to the current value of $x^{[i]}$, the value of $x^{[i]}$ at time ℓ , the

current value of $w^{[i]}$, and the messages received at time ℓ . This evolution model is very similar to the one that we introduced

for synchronous networks in Definition 1.40: in each communication round, the first step is transmission and the second one is computation and, except for the dependence on the physical state x , the communication and state transition processes are identical.

These ideas are formalized in the following definition.

Definition 3.10 (Evolution of a robotic network). Let $\mathbf{CC}$ be a control

and communication law for the robotic network S . The evolution of (S, CC) from initial conditions $x^{[i]} \in X^{[i]}$ and $w^{[i]} \in W^{[i]}, i \in I$, is the collection of curves $x^{[i]} : \mathbb{R}_{\geq 0} \rightarrow X^{[i]}$ and $w^{[i]} : \mathbb{Z}_{\geq 0} \rightarrow W^{[i]}, i \in I$, defined by

$$\dot{x}^{[i]}(t) = f^{[i]}(x^{[i]}(t), \text{ctl}^{[i]}(x^{[i]}(t), x^{[i]}(\lfloor t \rfloor), w^{[i]}(\lfloor t \rfloor), y^{[i]}(\lfloor t \rfloor)), \Sigma$$

where $\lfloor t \rfloor = \max\{\ell \in \mathbb{Z}_{\geq 0} \mid \ell < t\}$, and

$$w^{[i]}(\ell) = \text{stf}^{[i]}(x^{[i]}(\ell), w^{[i]}(\ell - 1), y^{[i]}(\ell)),$$

with $x^{[i]}(0) = x^{[i]}$ and $w^{[i]}(-1) = w^{[i]}, i \in I$. In the previous equations, $y^{[i]} : \mathbb{Z}_{\geq 0} \rightarrow A^n$ (describing the messages received by processor i) has components

$y^{[i]}(\ell)$, for $j \in I$, defined by

$$y^{[i]}(\ell) = \begin{cases} \text{msg}^{[j]}(x^{[j]}(\ell), w^{[j]}(\ell - 1), i), & \text{if } (j, i) \in E_{\text{cmm}} \\ \text{null}, & \text{otherwise.} \end{cases}$$

For convenience, we define $w(t) = w(\lfloor t \rfloor)$ for all $t \in \mathbb{R}_{\geq 0}$, and let $\mathbb{R}_{\geq 0} \ni$

$t \mapsto (x(t), w(t))$ denote the curves $x^{[i]}$ and $w^{[i]}$, for $i \in \{1, \dots, n\}$.

Remarks 3.11 (Simplifications of control and communication laws).

- (i) A control and communication law CC is *static* if the processor state set $W^{[i]}$ is a singleton for all $i \in I$. This means that there is no meaningful evolution of the processor state. In this case, CC can be described by a tuple $(A, \{\text{msg}^{[i]}\}_{i \in I}, \{\text{ctl}^{[i]}\}_{i \in I})$, with $\text{msg}^{[i]} : X^{[i]} \times I \rightarrow A$, and $\text{ctl}^{[i]} : X^{[i]} \times X^{[i]} \times A \rightarrow U$, for $i \in I$.
- (ii) A control and communication law is *data-sampled* if the control functions are independent of the current position of the robot and depend only upon the robot's position at the last sample time. Specifically, the control functions have the following property: given a processor state $w^{[i]} \in W^{[i]}$, an array of messages $y^{[i]} \in A^n$, a current state $x^{[i]}$, and a state at last sample time $x^{[i]}$, the

control

input $\text{ctl}^{[i]}(x^{[i]}, x_{\text{smpld}}^{[i]}, w^{[i]}, y^{[i]})$ is independent of $x^{[i]}$, for all $i \in I$.

In this case, the control functions can be described by maps of the form $\text{ctl}^{[i]} : X^{[i]} \times W^{[i]} \times \mathbf{A}^n \rightarrow U^{[i]}$, for $i \in I$.

- (iii) In many control and communication laws, the robots exchange full information about their states, including both their processor and their physical states. For such laws, we identify the communication alphabet with $\mathbf{A} = (X \times W) \cup \{\text{null}\}$ and we refer to the corresponding message-generation function $\text{msg}_{\text{std}}(x, w, j) = (x, w)$ as the *standard message-generation function*.

Note that we allow the processor state set and the communication alphabet to contain an infinite number of symbols. In other words, we assume that a robot can store and transmit a (finite number of) integer and real numbers, among other things. This is equivalent to assuming that we neglect any inaccuracies due to quantization, as we did in Section 1.6.

Remark 3.12 (Extensions of control and communication laws). Here, we briefly discuss alternative models and extensions of the proposed models.

Asynchronous sensor-based interactions. In the early network model proposed by Suzuki and Yamashita (1999), robots are referred to as “anonymous” and “oblivious” in precisely the same way in which we defined control and communication laws to be uniform and static, respectively. As compared with our notion of robotic network, the model in Suzuki and Yamashita (1999) is more general in that the robots’ activation schedules do not necessarily coincide (i.e., this model is asynchronous), and at the same time it is less general in that (1) robots cannot communicate any information other than their respective positions, and (2) each robot observes every other robot’s position (i.e., the complete communication graph is adopted). In the Section 3.2 below, we present a model in which robots rely on sensing rather than communication for their interaction.

Discrete-time motion models. For some algorithms in later chapters, it will be convenient to consider discrete-time motion models; for example, we present discrete-time motion models for first-order agents in Section 4.1. In some other cases, it will be convenient to consider dynamical interactions between agents taking place in continuous time.

Stochastic link models. Although we do not present any results on this topic in this notes, it is possible to

develop robotic networks models over random graphs and random geometric graphs, as studied by Bollobás (2001) and Penrose (2003). Furthermore, it is of interest to consider communication links with time-varying rates.

.

3.1.3 The agree and pursue control and communication law

We conclude this section with an example of a dynamic control and communication law. The problem is described as follows: a collection of robots with range-limited communication are placed on the unit circle; the robots move and communicate with the objectives of (1) agreeing on a direction of motion (clockwise or counterclockwise) and (2) achieving an equidistant configuration where all robots are equally angularly spaced. To achieve these

two objectives, we combine ideas from leader election algorithms for syn-chronous networks (see Section 1.5.4) and from cyclic pursuit problems (see Exercise E1.30): the robots move a distance proportional to an appropriate inter-robot separation, and they repeatedly compare their identifiers to discover the direction of motion of the robot with the largest identifier. In other words, the robots run a leader election task in their processor states and a uniform robotic deployment task in their physical state—these are among the most basic tasks in distributed algorithms and cooperative control. We present the algorithm here and characterize its correctness and performance later in the chapter.

From Example 3.7, we consider the uniform network S_{circle} of locally connected first-order robots on S^1 . For $r, u_{\max}, k_{\text{prop}} \in]0, 1[$ with $k_{\text{prop}}r \leq u_{\max}$, we define the agree & pursue law, denoted by $\text{agree \& pursue}$, as the *uniform data-sampled* law loosely described as follows:

[Informal description] The processor state consists of dir (the robot's direction of motion) taking values in c, cc (meaning clockwise and counterclockwise) and max-id (the largest UID received by the robot, initially set to the robot's UID) taking values in I . In each communication round, each robot transmits its position and its processor state. Among the messages received from agents moving toward its position, each agent picks the message with the largest value of max-id . If this value is larger than its own value, the agent resets its processor state with the selected message. Between communication rounds, each robot moves in the clockwise or counterclockwise direction depending on whether its processor state dir is c or cc . Each

robot moves k_{prop} times the distance to the immediately next neighbor in the chosen direction, or, if no neighbors are detected, k_{prop} times the communication range r .

Note that the processor state with the largest UID will propagate through- out the network as in the floodmax algorithm for leader election. Also, note that the assumption $k_{\text{prop}}r \leq u_{\text{max}}$ guarantees that the desired control is always within the allowable range $[u_{\text{max}}, u_{\text{max}}]$. Next, we define the law formally:

Robotic Network: $\mathcal{S}$: circle, first-order agents in $\mathbf{S}^1$
with absolute sensing of own position, and
with communication range r

Distributed Algorithm: agree & pursue

Alphabet: $\mathbf{A} = \mathbf{S}^1 \times \{c, cc\} \times I \cup \{\text{null}\}$

Processor State: $w = (\text{dir}, \text{max-id})$, where
 $\text{dir} \in \{c, cc\}$, initially: $\text{dir}^{[i]}$ unspecified
 $\text{max-id} \in I$, initially: $\text{max-id}^{[i]} = i$ for all i

% Standard message-generation function

function $\text{msg}(\theta, w, i)$

1: return (θ, w)

function $\text{stf}(\theta, w, y)$

1: for each non-null message $(\theta_{\text{rcvd}}, (\text{dir}_{\text{rcvd}}, \text{max-id}_{\text{rcvd}}))$ in y
do
2: if $(\text{max-id}_{\text{rcvd}} > \text{max-id})$ AND $(\text{dist}_{cc}(\theta, \theta_{\text{rcvd}}) \leq r)$ AND
 $\text{dir}_{\text{rcvd}} =$
 $c)$ OR $(\text{dist}_c(\theta, \theta_{\text{rcvd}}) \leq r)$ AND $\text{dir}_{\text{rcvd}} = cc)$ then
3: new-dir := dir_{rcvd}
4: new-id := $\text{max-id}_{\text{rcvd}}$
5: return $(\text{new-dir}, \text{new-id})$

function $\text{ctl}(\theta_{\text{smpld}}, w, y)$

1: $d_{\text{tmp}} := r$
2: for each non-null message $(\theta_{\text{rcvd}}, (\text{dir}_{\text{rcvd}}, \text{max-id}_{\text{rcvd}}))$ in y
do
3: if $(\text{dir} = cc)$ AND $(\text{dist}_{cc}(\theta_{\text{smpld}}, \theta_{\text{rcvd}}) < d_{\text{tmp}})$ then
4: $d_{\text{tmp}} := \text{dist}_{cc}(\theta_{\text{smpld}}, \theta_{\text{rcvd}})$
5: $u_{\text{tmp}} := k_{\text{prop}} d_{\text{tmp}}$
6: if $(\text{dir} = c)$ AND $(\text{dist}_c(\theta_{\text{smpld}}, \theta_{\text{rcvd}}) < d_{\text{tmp}})$ then
7: $d_{\text{tmp}} := \text{dist}_c(\theta_{\text{smpld}}, \theta_{\text{rcvd}})$
8: $u_{\text{tmp}} := -k_{\text{prop}} d_{\text{tmp}}$
9: return u_{tmp}

An implementation of this control and communication law is shown in Figure 3.4. As parameters, we select $n = 45$, $r = 2\pi/40$, $u_{\text{max}} = 1/4$ and $k_{\text{prop}} = 7/16$. Along the evolution, all robots agree upon a common direction of motion and, after a suitable time, they reach a uniform distribution.



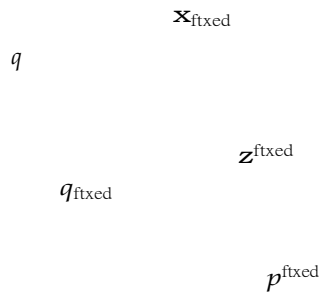
Figure 3.4 The agree & pursue law. Red-colored disks and blue-colored circles correspond to robots moving counterclockwise and clockwise, respectively. The initial positions and the initial directions of motion are randomly generated. The five diagrams depict the state of the network at times 0, 9, 20, 100, and 800.

3.2 ROBOTIC NETWORKS WITH RELATIVE SENSING

The model presented above assumes the ability of each robot to know its own absolute position. Here, we treat the alternative setting in which the robots do not communicate amongst themselves, but instead detect and measure each other's relative position through appropriate sensors. Additionally, we assume that the robots will perform measurements of the environment without having any *a priori* knowledge of it. We assume that robots do not have the ability to perform measurements expressed in a common reference frame. An early reference in which relative information is adopted is [Lin et al. \(2005\)](#).

3.2.1 Kinematics notions

Because the robots do not have a common reference frame, all the measurements generated by their on-board sensors are expressed in a local reference frame. To formalize this fact, it is useful to review some basic kinematics conventions. We let $\Sigma^{\text{fixed}} = (p^{\text{fixed}}, \mathbf{x}^{\text{fixed}}, y^{\text{fixed}}, z^{\text{fixed}})$ be a fixed reference frame in $\mathbb{R}^3$. A point q , a vector v , and a set of points S expressed with respect to the frame Σ^{fixed} are denoted by q^{fixed} , v^{fixed} and S^{fixed} , respectively. Next, let $\Sigma^b = (p^b, \{\mathbf{x}^b, y^b, z^b\})$ be a reference frame fixed to



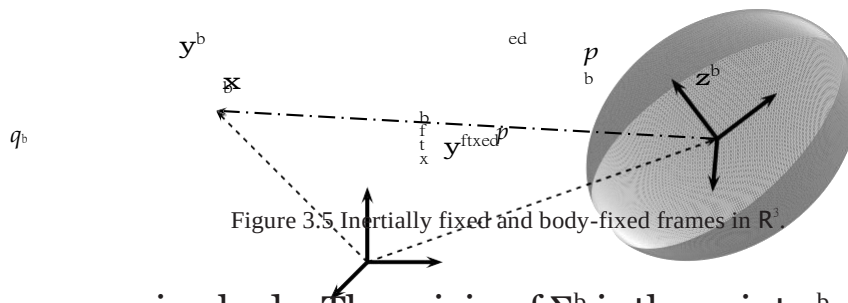


Figure 3.5. Inertially fixed and body-fixed frames in $\mathbb{R}^3$.

a moving body. The origin of Σ^b is the point p^b , fixed when denoted by p^b

expressed with respect to Σ^{fixed} . The orientation of Σ^b is characterized by the d -dimensional rotation R^b , whose columns are the frame vectors $\{x^b, y^b, z^b\}$ of Σ^b expressed with respect to Σ^{fixed} . We recall here the definition of the group of rotation matrices in d -dimensions:

$$SO(d) = \{R \in \mathbb{R}^{d \times d} \mid RR^T = I_d, \det(R) = +1\}.$$

With these notations, changes of reference frames are described by

$$\begin{aligned} q_{\text{fixed}} &= R_{\text{fixed}}^b q_b + p_{\text{fixed}}^b, \\ v_{\text{fixed}} &= R_{\text{fixed}}^b v_b, \\ S_{\text{fixed}} &= R_{\text{fixed}}^b S_b + p_{\text{fixed}}^b. \end{aligned} \quad (3.2.1)$$

Note that these change-of-frames formulas also hold in the planar case with the corresponding definition of the rotation matrix in $\text{SO}(2)$.

Remark 3.13 (Comparison with literature). In our notation, the sub-script denotes the frame with respect to which the quantity is expressed. Other references in the literature sometimes adopt the opposite convention, in which the superscript denotes the frame with respect to which the quantity is expressed.

•

3.2.2 Physical components

In what follows, we describe our notion of *mobile robots equipped with relative sensors*. We consider a group of n robots moving in an allowable environment $Q \subset \mathbb{R}^d$, for $d \in \{2, 3\}$, and we assume that a reference frame

$\Sigma^{[i]}$, for $i \in \{1, \dots, n\}$, is attached to each robot (see Figure 3.6). Expressed with respect to the fixed frame Σ^{fixed} , the i th frame $\Sigma^{[i]}$ is described by a position $p_{\text{fixed}}^{[i]} \in \mathbb{R}^d$ and an orientation $R^{[i]} \in \text{SO}(d)$. The continuous-time motion and discrete-time sensing models are described as follows.

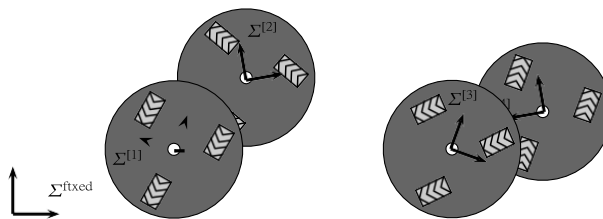


Figure 3.6 A robotic network with relative sensing. A group of four robots moves in $\mathbb{R}^2$.

Each robot $i \in \{1, \dots, 4\}$ has its own reference frame Σ^i .

Motion model: We select a simple motion model: for all $t \in \mathbb{R}_{\geq 0}$, the 150

orientation θ_i is constant in time and robot i translates according to

$$\dot{p}_i(t) = R_i \begin{bmatrix} \dot{p}_{\text{fixed}}^{[i]} \\ u^{[i]} \end{bmatrix}, \quad (3.2.2)$$

that is, the i th control input $u^{[i]}$ is known and applied in the robot frame. Each control input $u^{[i]}$, $i = 1, \dots, n$, takes values in a compact input space U . Clearly, it would be possible to consider a motion model with time-varying orientation and we refer the reader to Exercise E3.1, where we do so.

Sensing model: At each discrete time instant, robot i activates a sensor that detects the presence and returns a measurement about the relative position of any object (robots or environment boundary) inside a given “sensor footprint.” We describe the model in two steps. First, each robot measures

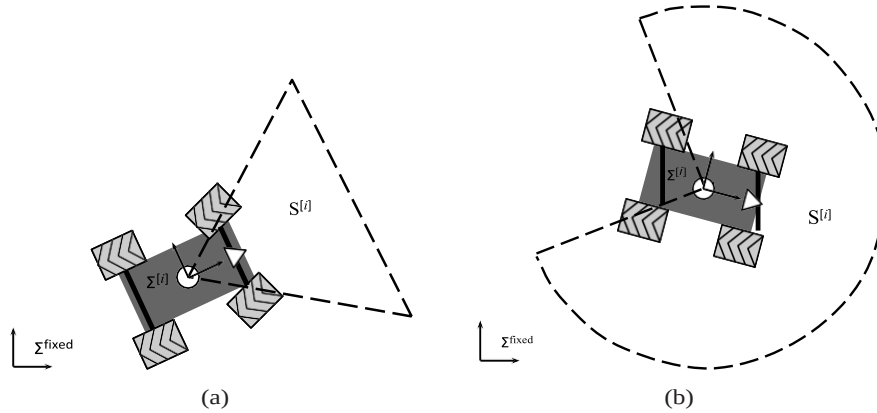


Figure 3.7 Examples of sensor footprints. (a) The cone-shaped sensor footprint of a vehicle equipped with a camera. (b) The 270-degree wedge-shaped sensor footprint of a vehicle equipped with a laser scanner.

other robots' positions and the environment as follows.

Sensing other robots' positions. There exists a set $\mathbf{A}_{\text{rht}}$ containing the null element, called the *sensing*

alphabet, and a map $\text{rbt-sns} : \mathbb{R}^d \rightarrow \mathbf{A}_{\text{rbt}}$, called the *sensing function*, with the interpretation that robot i acquires the symbol $\text{rbt-sns}(p^{[j]}) \in \mathbf{A}_{\text{rbt}}$ for each robot $j \in \{1, \dots, n\} \setminus \{i\}$.

Sensing the environment. There exists a set $\mathbf{A}_{\text{env}}$ containing the `null` element, called the *environment sensing alphabet*, and a map $\text{env-sns} : \mathcal{P}(\mathbb{R}^d) \rightarrow \mathbf{A}_{\text{env}}$, called the *environment sensing function*, with the interpretation that robot i acquires the symbol $\text{env-sns}(Q_i) \in \mathbf{A}_{\text{env}}$.

Second, we let $S^{[i]} \subset \mathbb{R}^d$ be the *sensor footprint* of robot i and we let

$S^{[i]}$ be its expression in the frame $\Sigma^{[i]}$ (see Figure 3.7). For simplicity, we

assume that all robot sensors are equal, so that we can write $S^{[i]} = S$. We require both sensing functions to provide no information about robots and boundaries that are outside S in the following two meanings: (i) if p is any point outside S , then $\text{rbt-sns}(p) = \text{null}$; and (ii) if W is any subset of $\mathbb{R}^d$, $\text{env-sns}(W) = \text{env-sns}(W \cap S)$.

We summarize this discussion with the following definition.

Definition 3.14 (Network with relative sensing). The physical components of a *network with relative sensing* consist of n mobile robots with identifiers $\{1, \dots, n\}$, with configurations in $Q \times \text{SO}(d)$, for an allowable en-

vironment $Q \subset \mathbb{R}^d$, with dynamics described by equation (3.2.2), and with relative sensors described by the sensor footprint S , sensing alphabets $\mathbf{A}_{\text{rbt}}$ and $\mathbf{A}_{\text{env}}$, and sensing functions rbt-sns and env-sns .

•

To make things concrete, let us present two examples of robotic networks with relative sensing that are analogs of the “communication-based” robotic networks S_{disk} and $S_{\text{vis-disk}}$ in Examples 3.4 and 3.6.

Example 3.15 (Disk sensor and corresponding relative-sensing net-

work). Given a *sensing range* $r \in \mathbb{R}_{>0}$, the *disk sensor* has sensor footprint

$B(0_d, r)$, that is, a disk sensor measures any object (robot and environment boundary) within distance r . Regarding sensing of other robots, we assume

that the alphabet is $\mathbf{A}_{\text{rbt}} = \mathbb{R}^d \cup \{\text{null}\}$ and that the sensing function is $\text{rbt-sns}(p^{[j]}) = p^{[j]}$ for each robot $j \in \{1, \dots, n\} \setminus \{i\}$, inside the sensor foot-

print $B(0_d, r)$, and $\text{rbt-sns}(p^{[j]}) = \text{null}$, otherwise. Regarding sensing of the environment, we assume that the alphabet is $A_{\text{env}} = P(R^d)$ and that the sensing function is $\text{env-sns}(Q_i) = Q_i \cap B(0_d, r)$. A group of robots with disk sensors defines the robotic network with relative sensing S^{rs} .

Example 3.16 (Range-limited visibility sensor and corresponding relative-sensing network). Given a sensing range $r \in R_{>0}$, the *range-limited visibility sensor* has sensor footprint $B(0_d, r)$ and performs measurements only of objects within unobstructed line of sight. Regarding sensing of other robots, we assume that the alphabet is $A_{\text{rbt}} = R^d \cup \{\text{null}\}$ and that the sensing function is $\text{rbt-sns}(p^{[j]}) = p^{[j]}$ for each robot $j \in \{1, \dots, n\} \setminus \{i\}$, inside the range-limited visibility set $\text{Vi}_{\text{disk}}(0_d; Q_i)$, and $\text{rbt-sns}(p^{[j]}) = \text{null}$, otherwise. Regarding sensing of the environment, we assume¹ that the alphabet is $A_{\text{env}} = P(R^d)$ and that the environment sensor measures the range-

¹It would be equivalent to assume that the robot can sense every portion of ∂Q that is within distance r and that is visible from the robot's position.

limited visibility set $\text{Vi}_{\text{disk}}(p^{[i]}; Q)$ expressed with respect to the frame $\Sigma^{[i]}$;

for the definition of range-limited visibility set, see Section 2.1.2. In other words, the environment sensing function is $\text{env-sns}(Q_i) = \text{Vi}_{\text{disk}}(0_2; Q_i)$. This is illustrated in Figure 3.8. A group of robots with range-limited visi-

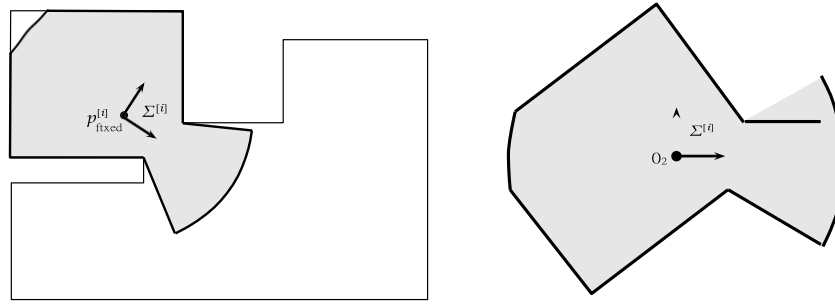


Figure 3.8 The left-hand plot depicts the range-limited visibility set $\text{Vi}_{\text{disk}}(p^{[i]}; Q)$ expressed with respect to an inertially fixed frame. The right-hand plot de-

presses with respect to the body-fixed frame $\Sigma^{[i]}$, that is, $\text{Vi}_{\text{disk}}(0_2; Q_i)$.

bility sensors defines the robotic network with relative sensing $\mathbf{S}^{\text{rs}}_{\text{vis-disk}}$.

Remark 3.17 (Sensing model consequences). The proposed sensing model has the following two consequences:

- (i) Robots have no information about the absolute position and orientation of themselves, the other robots or any part of the environment.
- (ii) The relative sensing capacity of the robots gives rise to a proximity graph, called the *sensing graph*, whose edges are the collection of robot pairs that are within sensing range. For example, in the network $\mathbf{S}^{\text{rs}}_{\text{disk}}$, the sensing graph is the disk graph $\mathbf{G}_{\text{disk}}(r)$. In general, sensing graphs are directed.

3.2.3 Relative-sensing control laws

As we did for robotic networks with interactions based on communication, we define here control laws based on relative sensing and we describe the closed-loop evolution of robotic networks with relative sensing.

First, we consider a robotic network with relative sensing S^{rs} characterized by: identifiers $\{1, \dots, n\}$, configurations in $Q \times \text{SO}(d)$, for an allowable environment $Q \subset \mathbb{R}^d$, dynamics described by equation (3.2.2), and relative sensors described by the sensor footprint S , sensing alphabets A_{rbi} and

A_{env} , and sensing functions rbt-sns and env-sns . A **RSC** *relative-sensing control law* for the robotic network with relative sensing $^{\text{rs}}$ consists of the following tuple:

- (i) W , called the *processor state set*, with a corresponding set of *allowable initial values* $W_0 \subseteq W$;
- (ii) $\text{stf} : W \times A_{\text{env}} \rightarrow W$, called the *(processor) state-transition function*; and $\times A_{\text{env}} \rightarrow U$, called the *(motion) control function*.
- (iii) $\text{ctl} : W \times A_{\text{rbt}}^n$

As for robotic networks, we say that RSC is *static* if W is a singleton for all $i \in \{1, \dots, n\}$; in this case, RSC can be described by a motion control function $\text{ctl} : A_{\text{rbt}}^n \times A_{\text{env}} \rightarrow U$. Additionally, if the environment $Q = \mathbb{R}^d$, then RSC can be described by a motion control function $\text{ctl} : W \times A^n \rightarrow U$.

Second, the *evolution* of $(S^{\text{rs}}, \text{RSC})$ from initial conditions $(p^{[i]}, R^{[i]}) \in \mathbb{R}^d \times \text{SO}(d)$ and $w^{[i]} \in W_0$, $i \in \{1, \dots, n\}$ is the collection of curves $p^{[i]} : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^d$ and $w^{[i]} : \mathbb{Z}_{\geq 0} \rightarrow W$, $i \in \{1, \dots, n\}$, defined by

$$\dot{p}^{[i]}(t) = R^{[i]}(t) \cdot w^{[i]}(t), \quad \dot{w}^{[i]}(t) = \text{ctl}(w^{[i]}(t), y^{[1]}(t), \dots, y^{[n]}(t), y^{\text{env}}(t)),$$

with $p^{[i]}(0) = p^{[i]}$ and $w^{[i]}(-1) = w^{[i]}$, $i \in \{1, \dots, n\}$. In the previous

equations, $y^{[i]} : \mathbb{Z}_{\geq 0} \rightarrow A^n$ (describing the robot measurements taken by

sensor i) with components $y^{[j]}(\ell)$, for $j \in \{1, \dots, n\}$, and $y^{\text{env}} : \mathbb{Z}_{\geq 0} \rightarrow A_{\text{env}}$

(describing the environment measurements taken by sensor i) are defined by

$$y^{[j]}(\ell) = \text{rbt-sns}(p^{[j]}(\ell)), \quad y^{\text{env}}(\ell) = \text{env-sns}(Q_i(\ell)).$$

In the last equation, $p^{[j]}$ and $Q(\ell)$ denote the position of the j -th robot and the environment Q as expressed with respect to the moving frame $\Sigma^{[j]}$.

3.2.4 Equivalence between control and communication laws and relative-sensing control laws

Consider a “communication-based” robotic network S_1 with a control and communication law CC_1 with the following properties:

- (i) Regarding S : the network is uniform, the state space is $X = \mathbb{R}^d$ with states denoted by $x^{[j]} = p^{[j]}$, the communication graph is the r -disk graph, and the robot dynamics are $\dot{p}^{[j]} = u^{[j]}$.
- (ii) Regarding CC_1 : the control and communication law is uniform and

data-sampled, the communication alphabet is $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$, and the message-generation function is $\text{msg}(p, w, j) = p$.

Given a network and a law (S_1, CC_1) satisfying (i) and (ii), the control and communication law CC_1 is *invariant* if its state transition and control maps satisfy, for all $p \in \mathbb{R}^d$, $w \in W$, $y \in \mathbf{A}^n$, and $R \in \text{SO}(d)$,

$$\begin{aligned} \text{stf}(p, w, y) &= \text{stf}(0_d, w, R(y - p))^\Sigma, \\ \text{ctl}(p, w, y) &= R^\top \text{ctl}(0_d, w, R(y - p))^\Sigma, \end{aligned}$$

where the i th component of $R(y - p) \in \mathbf{A}^n$ is $R(y_i - p)$ if $y_i \in \mathbb{R}^d$, or null if $y_i = \text{null}$.

Next, consider a relative-sensing network S_2 with disk sensors as in Example 3.15, that is, assume that the sensing footprint is $B(0_d, r)$, the sensing alphabet is $\mathbf{A}_{\text{rft}} = \mathbb{R}^d \cup \{\text{null}\}$, and the sensing function equals the identity function in $B(0_d, r)$. We assume no environment sensing as we set $Q = \mathbb{R}^d$. The communication and control law CC_1 and the relative-sensing control law RSC_2 for network S_2 are *equivalent* if their processor state sets identical, for example, denoting both by W , and their state transition and control maps

satisfy, for all $w \in W$ and $y \in \mathbb{R}^d \cup \{\text{null}\} = \mathbf{A}^n = \mathbf{A}_{\text{rft}}$

$$\text{stf}_1(0_d, w, y) = \text{stf}_2(w, y), \quad \text{and} \quad \text{ctl}_1(0_d, w, y) = \text{ctl}_2(w, y).$$

Proposition 3.18 (Evolution equivalence). *If CC_1 is invariant and if CC_1 and RSC_2 are equivalent, then the evolutions of the control and communication laws S_1, CC_1 and S_2, RSC_2 from identical initial conditions are identical.*

Proof. Assume that the messages and measurements array $y^{[i]}(t)$ received by the i -th robot at time t in the

communication-based network and in the relative-sensing networks are equal to, respectively:

$$p^{[j_1]}, \dots, p^{[j_k]}, \quad \text{and} \quad p^{[j_1]}, \dots, p^{[j_k]}.$$

$$p_{\text{fixed}}^{[j]} = \text{ctl}_1(p_{\text{fixed}}^{[j_1]}, \dots, p_{\text{fixed}}^{[j_k]}),$$

Under this assumption, the evolutions of the communication-based network and of the relative-sensing networks are written, respectively, as,

$$p_{\text{fixed}}^{[j]} = \text{ctl}_1(p_{\text{fixed}}^{[j_1]}, \dots, p_{\text{fixed}}^{[j_k]}),$$

$$p^{[j]} = R_{\text{fixed}}^{[j]} \text{ctl}_2(w^{[j]}, p^{[j_1]}, \dots, p^{[j_k]}).$$

$$p_{\text{fixed}}^{[j]} = R_{\text{fixed}}^{[j]} \text{ctl}_2(w^{[j]}, p_{\text{fixed}}^{[j_1]}, \dots, p_{\text{fixed}}^{[j_k]}).$$

From equation (3.2.1), we know that, for all $j \in \{j_1, \dots, j_k\}$,

$$p_{\text{fixed}}^{[j]} = R_{\text{fixed}}^{[j]} p_i^{[j]} + p_{\text{fixed}}^{[j]} \implies p_i^{[j]} = (R_{\text{fixed}}^{[j]})^T (p_{\text{fixed}}^{[j]} - p_{\text{fixed}}^{[i]}).$$

From this equality and from the fact that CC_1 is invariant, we observe that

$$\text{ctl}_1(\underbrace{p^{[i]}}_{\text{fixed}}, \underbrace{w^{[i]}}_{\text{fixed}}, \underbrace{p^{[j_1]}}_{\text{fixed}}, \dots, \underbrace{p^{[j_k]}}_{\text{fixed}}) = R^{[i]} \quad \text{ctl}_1(0_d, \underbrace{w^{[i]}}_i, \underbrace{p^{[j_1]}}_i, \dots, \underbrace{p^{[j_k]}}_i).$$

Since CC_1 and CC_2 are equivalent, the two evolution equations coincide. A similar reasoning also shows that the evolutions of the processor states are identical.

□

Remark 3.19 (Communication-based laws on relative-sensing networks). Proposition 3.18 implies the following fact. An invariant control and communication law for a robotic network satisfying appropriate properties can be implemented on an appropriate relative-sensing network as a relative-sensing control law. •

3.3 COORDINATION TASKS AND COMPLEXITY NOTIONS

In this section, we introduce concepts and tools that are useful analyzing a communication and control law in a robotic network; our treatment is directly generalized to relative-sensing networks. We address the following questions: What is a coordination task for a robotic network? When does a control and communication law achieve a task? And with what time, space, and communication complexity?

3.3.1 Coordination tasks

Our first analysis step is to characterize the correctness properties of a communication and control law. We do so by defining the notions of task and of task achievement by a robotic network.

Definition 3.20 (Coordination task). Let S be a robotic network and let W be a set.

- (i) A *coordination task* is a map $T : \prod_{i \in I} X_i \times W \rightarrow \{\text{true}, \text{false}\}$.
- (ii) If W is a singleton, then the coordination task is said to be *static*

and can be described by a map $T : \prod_{i \in I} X_i \rightarrow \{\text{true}, \text{false}\}$.

Additionally, let CC be a control and communication law for S :

- (i) The law CC is *compatible* with the task $T : \prod_{i \in I} X_i \times W \rightarrow \{\text{true}, \text{false}\}$ if its processor state takes values in W , that is, if $W^{[i]} = W$, for all $i \in I$.
- (ii) The law CC *achieves* the task T if it is compatible with it and if, for all initial conditions $x_0^{[i]} \in X_0^{[i]}$ and $w_0^{[i]} \in W_0^{[i]}$ $i \in I$, there exists

$T \in \mathbb{R}_{>0}$ such that the network evolution $t \mapsto (x(t), w(t))$ has the property that $T(x(t), w(t)) = \text{true}$ for all $t \geq T$.

•

Remark 3.21 (Temporal logic). Loosely speaking, the phrase “a law achieves a task” means that the network evolutions reach (and remain at) a specified pattern in the robot physical or processor state. In other words, the task is achieved if *at some time and for all subsequent times*, the predicate evaluates to `true` along system trajectories. It is possible to consider more general tasks based on more expressive predicates on trajectories. Such predicates can be defined through various forms of temporal and propositional logic, (see, e.g., [Emerson, 1994](#)). In particular, (linear) temporal logic contains certain constructs that allow reasoning in terms of time and is hence appropriate for robotic applications—as argued, for example, by [Fainekos et al. \(2005\)](#). Network tasks such as periodically visiting a desired set of configurations can be encoded with temporal logic statements.

•

Example 3.22 (Direction agreement and equidistance tasks). From Example 3.7, consider the uniform network of locally connected first-order agents in S^1 . From Section 3.1.3, recall the agree & pursue control and communication law $\text{CC}_{\text{agree \& pursue}}$ with processor state taking values in $W = \{\text{cc}, \text{c}\} \times I$. There are two tasks of interest. First, we define the *direction agreement task* $T_{\text{dir}} : (S^1)^n \times W^n \rightarrow \{\text{true}, \text{false}\}$ by

$$T_{\text{dir}}(\theta, w) = \begin{cases} \text{true}, & \text{if } \text{dir}^{[1]} = \dots = \text{dir}^{[n]}, \\ \text{false}, & \text{otherwise} \end{cases}$$

where $\theta = (\theta^{[1]}, \dots, \theta^{[n]})$, $w = (w^{[1]}, \dots, w^{[n]})$, and $w^{[i]} = (\text{dir}^{[i]}, \text{max-id}^{[i]})$, for $i \in I$. Furthermore, for $\varepsilon > 0$, we define the static (agent) equidistance task $T_{\varepsilon\text{-eqdstnc}} : (S^1)^n \rightarrow \{\text{true}, \text{false}\}$ to be true if and only if

$$\min_{j \neq i} \text{dist}_{\text{cc}}(\theta^{[i]}, \theta^{[j]}) \cdot \min_{j \neq i} \text{dist}_{\text{cc}}(\theta^{[i]}, \theta^{[j]}) < \varepsilon, \quad \text{for all } i \in I.$$

In other words, $\varepsilon\text{-eqdstnc}$ is true when, for every agent, the

distances to the closest clockwise neighbor and to the closest counterclockwise neighbor are approximately equal.

•

3.3.2 Complexity notions

We are now ready to define the notions of time, space and communication complexity. These notions describe the cost that a certain control and communication law incurs while completing a certain coordination task. Additionally, the complexity of a task is the infimum of the costs incurred by all laws that achieve that task. We begin by highlighting a difference between what follows and the complexity treatment for synchronous networks.

Remark 3.23 (Termination via task completion). As discussed in Remark 1.44 in Section 1.5, it is possible to consider various algorithm termination notions. Here, we will establish the completion of an algorithm as the instant when a given task is achieved.

.

First, we define the time complexity of an achievable task as the minimum number of communication rounds needed by the agents to achieve the task T .

Definition 3.24 (Time complexity). Let S be a robotic network and let T be a coordination task for S . Let CC be a control and communication law for S compatible with T :

- (i) the (worst-case) time complexity to achieve T with CC from initial conditions $(x_0, w_0) \in \prod_{i \in I} X_0 \times \prod_{i \in I} W_0$ is

$$TC(T, CC, x_0, w_0) = \inf \{ \ell \mid T(x(k), w(k)) = \text{true}, \text{ for all } k \geq \ell \},$$

where $t \mapsto (x(t), w(t))$ is the evolution of (S, CC) from the initial condition (x_0, w_0) ;

- (ii) the (worst-case) time complexity to achieve T with CC is

$$TC(T, CC) = \sup_{(x_0, w_0) \in \prod_{i \in I} X_0 \times \prod_{i \in I} W_0} TC(T, CC, x_0, w_0);$$

- (iii) the (worst-case) time complexity of T is

$$TC(T) = \inf \{ TC(T, CC) \mid CC \text{ compatible with } T \}.$$

.

Next, we quantify memory and communication requirements of communication and control laws. We assume that elements of the processor state set W or of the alphabet set A might amount to multiple “basic memory

units” or “basic messages.” We let W_{basic} and A_{basic} denote the number of basic memory units and basic messages required to represent elements of W and A , respectively. The null message has zero cost. To clarify this assumption, we adopt two conventions. First, as in Section 1.5.2, we assume that a “basic memory unit” or a “basic message” contains $\log(n)$ bits. This implies that the $\log(n)$ bits required to store or transmit a robot identifier $i \in \{1, \dots, n\}$ are equivalent to one “basic memory unit.” Second, as mentioned in Remark 3.11, we assume that a processor can store and transmit a (finite number of) integer and real numbers, and we adopt the convention that any such number is quantized and represented by a constant number of basic memory units or basic messages.

We now quantify memory requirements of algorithms and tasks by count-

ing the required number of basic memory units. Let the network S , the task T , and the control and communication law CC be as in Definition 3.24.

Definition 3.25 (Space complexity).

- (i) The *(worst-case) space complexity to achieve T with CC* , denoted by $SC(S, T, CC)$, is the maximum number of basic memory units required by a robot processor executing the on among all robots and among all allowable initial physical and processor states until termination; and
- (ii) the space complexity of T is the infimum among the space complexities of all control and communication laws that achieve T .

The set of all non-null messages generated during one communication round from network state (x, w) is denoted by

$$M(x, w) = \{(i, j) \in E_{\text{comm}}(x) \mid \text{msg}(x_{[i]}, w_{[j]}) \neq \text{null}\}.$$

We now quantify the mean and total communication requirements of algorithms and tasks by counting the number of transmitted basic messages.

Definition 3.26 (Mean and Total Communication complexity).

- (i) The *(worst-case) mean communication complexity* and the *(worst-case) total communication complexity* to achieve T with CC from $(x_0, w_0) \in \prod_{i \in I} X_0 \times \prod_{i \in I} W_0$ are, respectively,

$$\begin{aligned} MCC(T, CC, x_0, w_0) &= \frac{|A|_{\text{basic}}}{\tau} \sum_{\ell=0}^{\tau-1} |M(x(\ell), w(\ell))|, \\ TCC(T, CC, x_0, w_0) &= |A|_{\text{basic}} \sum_{\ell=0}^{\tau-1} |M(x(\ell), w(\ell))|, \end{aligned}$$

where $t \mapsto (x(t), w(t))$ is the evolution of (S, CC) from the initial condition (x_0, w_0) and where $\tau = TC(CC, T, x_0, w_0)$. Here, MCC is defined only for initial conditions (x_0, w_0) with the property that

$T(x_0, w_0) = \text{false};$

- (ii) the *(worst-case) mean communication complexity* (resp. the *(worst-case) total communication complexity*) to achieve T with CC is the supremum of $MCC(T, CC, x_0, w_0)$ (resp. $TCC(T, CC, x_0, w_0)$) over all allowable initial states (x_0, w_0) ; and
- (iii) the *(worst-case) mean communication complexity* (resp. the *worst-case total communication complexity*) of T is the infimum among the mean communication complexity (resp. the total communication complexity) of all control and communication laws achieving T . •

By construction, one can verify that it always happens that

$$\text{TCC}(\mathcal{T}, \mathcal{CC}) \leq \text{MCC}(\mathcal{T}, \mathcal{CC}) \cdot \text{TC}(\mathcal{T}, \mathcal{CC}). \quad (3.3.1)$$

We conclude this section with possible variations and extensions of the complexity definitions.

Remark 3.27 (Infinite-horizon mean communication complexity). The mean communication complexity MCC measures the average cost of the communication rounds required to achieve a task over a finite time horizon; a similar statement holds for the total communication complexity TCC. One might be interested in a notion of mean communication complexity required to maintain the task true for all times. Accordingly, the infinite-horizon mean communication complexity of $\mathcal{CC}$ from initial conditions (x_0, w_0) is

$$\text{IH-MCC}(\mathcal{CC}, x_0, w_0) = \lim_{\tau \rightarrow \infty} \frac{1}{\tau} \sum_{\ell=0}^{\tau} |M(x(\ell), w(\ell))|. \quad \bullet$$

Remark 3.28 (Communication complexity in omnidirectional networks). In omnidirectional wireless networks, the standard operation mode is for all neighbors of a node to receive the signal that it transmits. In other words, the transmission is omnidirectional rather than unidirectional. It is straightforward to require the message-generation function to have the property that the output it generates be independent of the intended receiver. Under such assumptions, it makes sense to count as communication complexity not the number of messages transmitted in the network, but the number of transmissions, that is, a unit cost per node rather than a unit cost per edge of the network. •

Remark 3.29 (Energy complexity). Given a model for the energy consumed by the robot to move and to transmit a message, one can easily define a notion of energy complexity for a control and communication law. In modern wireless transmitters, the energy consumption in transmitting a signal at a distance r varies with a power

of r . Analogously, energy consumption is an increasing function of distance traveled. We consider this to be a promising avenue for further research.

•

3.3.3 Invariance under rescheduling

Here we discuss the invariance properties of time and communication complexity under the *rescheduling* of a control and communication law. The idea behind rescheduling is to “spread” the execution of the law over time without affecting the trajectories described by the robots.

For simplicity we consider the setting of static laws; similar results can be obtained for the general setting. Also, for ease of presentation, we allow our communication and control laws to be time dependent, that is, we consider message-generation functions and motion control functions of the form $\text{msg}^{[i]} : \mathbb{Z}_{\geq 0} \times X^{[i]} \times I \rightarrow A$ and $\text{ctl}^{[i]} : \mathbb{R}_{\geq 0} \times X^{[i]} \times X^{[i]} \rightarrow A^n \cup \{0\}$, respectively. Definition 3.10 for network evolution can be readily extended to this more general time-dependent setup.

Let $S = (I, R, E_{\text{cmm}})$ be a robotic network in which each robot is a driftless control system (see Section 1.3). Let $\mathcal{C} = (A, \text{msg}^{[i]}_{i \in I}, \text{ctl}^{[i]}_{i \in I})$ be a static control and communication law. In what follows, we define a new control and communication law by modifying $\mathcal{C}$; to do so, we introduce some notation. Let $s \in \mathbb{N}$, with $s \leq n$, and let $P_I = \{I_0, \dots, I_{s-1}\}$ be an s -partition of I , that is, $I_0, \dots, I_{s-1} \subset I$ are disjoint and nonempty and $I = \bigcup_{k=0}^{s-1} I_k$. For $i \in I$, define the message-generation functions

$$\begin{aligned} \text{msg}^{[i]}_{P_I} : \mathbb{Z}_{\geq 0} \times X^{[i]} \times I &\rightarrow A \text{ by} \\ \text{msg}^{[i]}_{P_I}(\ell, x, j) &= \text{msg}^{[i]}(\lfloor \ell/s \rfloor, x, j), \end{aligned} \quad (3.3.2)$$

if $i \in I_k$ and $k = \ell \bmod s$, and $\text{msg}^{[i]}_{P_I}(\ell, x, j) = \text{null}$ otherwise. According to this message-generation function, only the agents with a unique identifier in I_k will send messages at time ℓ , where $\ell = k + as$, $a \in \mathbb{Z}_{\geq 0}$. Equivalently, this can be stated as follows: according to (3.3.2), the messages originally sent at the time instant ℓ are now rescheduled to be sent at the time instants $F(\ell) - s + 1, \dots, F(\ell)$, where $F : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{Z}_{\geq 0}$ is defined by $F(\ell) = s(\ell + 1) - 1$. Figure 3.9 illustrates this idea. For $i \in I$, define the control functions

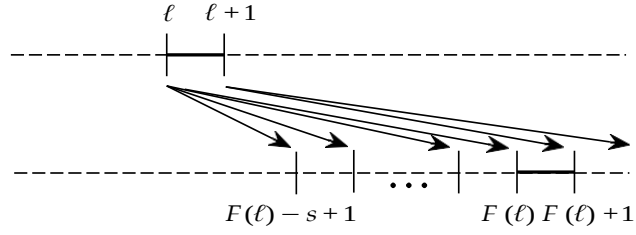


Figure 3.9 Under the rescheduling, the messages that are sent at the time instant ℓ under the control and communication law CC are rescheduled to be sent over the time instants $F(\ell) - s + 1, \dots, F(\ell)$ under the control and communication law $\text{CC}_{(s, \mathbf{P}_f)}$.

$\text{ctl}^{[i]}: \mathbb{R}_{\geq 0} \times X^{[i]} \times X^{[i]} \times \mathbf{A}^n \rightarrow U^{[i]}$ by

$$\text{ctl}_{\mathbf{P}_f}^{[i]}(t, x, x_{\text{smpld}}, y) = \text{ctl}^{[i]} \left(t - \ell + F^{-1}(\ell), x, x_{\text{smpld}}, y \right) \Sigma, \quad (3.3.3)$$

if $t \in [\ell, \ell + 1]$ and $\ell = -1 \bmod s$, and $\text{ctl}_{\mathbf{P}_f}^{[i]}(t, x, x_{\text{smpld}}, y) = 0$ otherwise.

Here, $F^{-1} : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{Z}_{\geq 0}$ is the inverse of F , defined by $F^{-1}(\ell) = \lfloor \frac{\ell+1}{s} \rfloor - 1$.

Roughly speaking, the control law $\text{ctl}^{[i]}$ makes the agent i wait for the time intervals $[\ell, \ell + 1]$, with $\ell \in \{as - 1\}$

Accordingly, the evolution of the robotic network under the original law ρ during the time interval $[\ell, \ell + 1]$ now takes place when all the corresponding messages have been transmitted, that is, along the time interval $[F(\ell), F(\ell) + 1]$. The following definition summarizes this construction.

Definition 3.30 (Rescheduling of control and communication laws). Let $S = (I, R, E_{\text{cmm}})$ be a robotic network with driftless physical agents, and let $\text{CC} = (\mathbb{Z}_{\geq 0}, \mathbf{A}, \{\text{msg}^{[i]}\}_{i \in I}, \{\text{ctl}^{[i]}\}_{i \in I})$ be a static control and communication law. Let $s \in \mathbb{N}$, with $s \leq n$, and let P_I be an s -partition of I . The control and communication law $\text{CC}_{(s, P_I)} = (\mathbb{Z}_{\geq 0}, \mathbf{A}, \{\text{msg}^{[i]}\}_{i \in I}, \{\text{ctl}^{[i]}\}_{i \in I})$ defined by equations (3.3.2) and (3.3.3) is called a P_I -rescheduling of CC .

The following result, whose proof is presented in Section 3.6.1, shows that the total communication complexity is invariant under rescheduling.

Proposition 3.31 (Complexity of rescheduled laws). *With the assumptions of Definition 3.30, let $T : \prod_{i \in I} X^{[i]} \rightarrow \{\text{true}, \text{false}\}$ be a coordination task for S . Then, for all $x_0 \in \prod_{i \in I} X_0$,*

$$\text{TC}(T, \text{CC}_{(s, P_I)}, x_0) = s \cdot \text{TC}(T, \text{CC}, x_0).$$

Moreover, if C_{md} is additive, then, for all $x_0 \in \prod_{i \in I} X_0^{[i]}$,

1

$$\text{MCC}(\mathbf{T}, \text{CC}_{(S, P, \tau)}, x_0) = \frac{1}{S} \cdot \text{MCC}(\mathbf{T}, \text{CC}, x_0),$$

and therefore, $\text{TCC}(\mathbf{T}, \text{CC}_{(S, P, \tau)}, x_0) = \text{TCC}(\mathbf{T}, \text{CC}, x_0)$, that is, the total communication complexity of CC is invariant under rescheduling.

Remark 3.32 (Appropriate complexity notions for driftless agents). Given the results in the previous theorem, one should be careful in choosing which notion of communication complexity to use in order to evaluate control and communication laws. For driftless physical agents, rather than the *mean* communication complexity MCC , one should really consider the *total* communication complexity TCC , since the latter is invariant with respect to rescheduling. Note that the notion of infinite-horizon mean communication complexity IH-MCC defined in Remark 3.27 satisfies the same relationship as MCC , that is, $\text{IH-MCC}(\text{CC}_{(S, P)}, x_0) = \frac{1}{S} \cdot \text{IH-MCC}(\text{CC}, x_0)$.

•

3.4 COMPLEXITY OF DIRECTION AGREEMENT AND EQUIDISTANCE

From Example 3.7, Section 3.1.3, and Example 3.22, recall the definition of a uniform network S_{circle} of locally connected first-order agents in S^1 , the agree & pursue control and communication law $CC_{\text{agree \& pursue}}$, and the two coordination tasks T_{dir} and $T_{\varepsilon\text{-eqdstnc}}$. In this section, we characterize the complexity to achieve these coordination tasks with $CC_{\text{agree \& pursue}}$. Because the number of bits required to represent the variable $\text{max-id} \in \{1, \dots, n\}$ is $\log(n)$, note that the space complexity of $CC_{\text{agree \& pursue}}$ is $\log(n)$ bits, that is, one basic memory unit in our convention discussed in Section 3.3.2.

Motivated by Remark 3.8, we model wireless communication congestion by assuming that the communication range is a monotone non-increasing function $r : \mathbb{N} \rightarrow]0, \pi[$ of the number of agents n . Likewise, we assume that the maximum control amplitude u_{max} is a non-increasing function $u_{\text{max}} : \mathbb{N} \rightarrow]0, 1[$; recall that u_{max} is the maximum robot speed. Finally, it is convenient to define the function $\delta(n) = nr(n) - 2\pi$ that compares the sum of the communication ranges of all the robots with the length of the unit circle.

We are now ready to state the main result of this section; proofs are postponed to Section 3.6.2.

Theorem 3.33 (Time complexity of agree-and-pursue law). *Given $k_{\text{prop}} \in]0, 1[$, in the limit as $n \rightarrow +\infty$ and $\varepsilon \rightarrow 0$, the network S_{circle} with $u_{\text{max}}(n) \geq k_{\text{prop}}r(n)$, the law $CC_{\text{agree \& pursue}}$, and the tasks T_{dir} and $T_{\varepsilon\text{-eqdstnc}}$ together satisfy the following properties:*

- (i) $TC(T_{\text{dir}}, CC_{\text{agree \& pursue}}) \in \Theta(r(n)^{-1})$.
- (ii) If $\delta(n)$ is lower bounded by a positive constant as $n \rightarrow +\infty$, then

$$\begin{aligned} \text{TC}(\mathcal{T}_{\varepsilon\text{-eqdstnc}}, \mathcal{CC}_{\text{agree \& pursue}}) &\in \Omega(n \log(n\varepsilon^{-1})), \\ \text{TC}(\mathcal{T}_{\varepsilon\text{-eqdstnc}}, \mathcal{CC}_{\text{agree \& pursue}}) &\in O(n \log(n\varepsilon^{-1})). \end{aligned}$$

If $\delta(n)$ is upper bounded by a negative constant, then in general the law $\mathcal{CC}_{\text{agree \& pursue}}$ does not achieve $\mathcal{T}_{\varepsilon\text{-eqdstnc}}$.

Next, we study the total communication complexity of the agree-and- pursue control and communication law.

First, we note that any message in $\mathbf{A} = \mathcal{S}^1 \times \{\text{cc}, \text{c}\} \times \{1, \dots, n\} \cup \{\text{null}\}$ requires only a finite number of basic messages to encode, that is, $|\mathbf{A}|_{\text{basic}} \in O(1)$.

Theorem 3.34 (Total communication complexity of agree-and-pur-

sue law). For $k_{\text{prop}} \in]0, 1[$, in the limit as $n \rightarrow +\infty$ and $\varepsilon \rightarrow 0^+$, the network S_{circle} with $u_{\text{max}}(n) \geq k_{\text{prop}}r(n)$, the law $\text{CC}_{\text{agree \& pursue}}$, and the tasks T_{dir} and $T_{\varepsilon\text{-eqdstnc}}$ together satisfy the following properties:

(i) If $\delta(n) \geq \pi(1/k_{\text{prop}} - 2)$ as $n \rightarrow +\infty$, then

$$\text{TCC}(T_{\text{dir}}, \text{CC}_{\text{agree \& pursue}}) \in \Theta(n^2 r(n)^{-1});$$

otherwise, if $\delta(n) \leq \pi(1/k_{\text{prop}} - 2)$ as $n \rightarrow +\infty$, then

$$\begin{aligned} \text{TCC}(T_{\text{dir}}, \text{CC}_{\text{agree \& pursue}}) &\in \Omega(n^3 + nr(n)^{-1}), \text{TCC}(T_{\text{dir}}, \\ \text{CC}_{\text{agree \& pursue}}) &\in O(n^2 r(n)^{-1}). \end{aligned}$$

(ii) If $\delta(n)$ is lower bounded by a positive constant as $n \rightarrow +\infty$, then

$$\begin{aligned} \text{TCC}(T_{\varepsilon\text{-eqdstnc}}, \text{CC}_{\text{agree \& pursue}}) &\in \Omega(n^3 \delta(n) \log(n\varepsilon)^{-1}), \text{TCC}(T_{\varepsilon\text{-eqdstnc}}, \\ \text{CC}_{\text{agree \& pursue}}) &\in O(n^4 \log(n\varepsilon^{-1})). \end{aligned}$$

Remark 3.35 (Comparison with leader election). Let us compare the agree-and-pursue control and communication law with the classical Le Lann-Chang-Roberts (LCR) algorithm for leader election discussed in Section 1.5.4. The leader election task consists of electing a unique agent among all agents in the network; therefore, it is different from, but closely related to, the coordination task dir . The LCR algorithm operates on a static network with the ring communication topology, and achieves leader election with time and total communication complexity $\Theta(n)$ and $\Theta(n^2)$, respectively. The agree-and-pursue law operates on a robotic network with the $r(n)$ -disk communication topology, and achieves dir with time

and total communication complexity, respectively, $\Theta(r(n)^{-1})$ and $O(n^2 r(n)^{-1})$. If wireless communication congestion is modeled by $r(n)$ of order $1/n$ as in Remark 3.8, then the two algorithms have identical time complexity and the LCR algorithm has better communication complexity. Note that computations on a possibly disconnected, dynamic network are more complex than on a static ring topology.

•

3.5 NOTES

The study of multi-robot systems has a long and rich history. Some recent examples include the surveys (Asama, 1992; Cao et al., 1997; Dias et al., 2006), the text by Arkin (1998) on behavior-based robotics, and the special issues (Arai et al., 2002; Abdallah and Tanner, 2007; Bullo et al., 2009). Together with this literature, the starting points for developing the material in this chapter are the standard notions of *synchronous and asynchronous networks* in distributed (Lynch, 1997; Peleg, 2000; Tel, 2001) and parallel (Bertsekas and Tsitsiklis, 1997; Parhami, 1999) computation. The established body of knowledge on synchronous networks is, however, not directly

applicable to the robotic network setting because of the agents' mobility and the ensuing dynamic communication topology.

An influential contribution toward a network model of mobile interacting robots is the work by [Suzuki and Yamashita \(1999\)](#). This model consists of a group of identical “distributed anonymous mobile robots” characterized as follows: no explicit communication takes place between them, and at each time instant of an “activation schedule,” each robot senses the relative position of all other robots and moves according to a pre-specified algorithm. An artificial intelligence approach to multi-agent behavior in a shared environment is taken in [Moses and Tennenholtz \(1995\)](#). [Santoro \(2001\)](#) provides, with an emphasis on computer science aspects, a brief survey of models, algorithms, and the need for appropriate complexity notions. Recently, a notion of communication complexity for control and communication algorithms in multi-robot systems has been analyzed by [Klavins \(2003\)](#); see also [Klavins and Murray \(2004\)](#). Notions of failures and robustness in robotic networks are discussed by [Gupta et al. \(2006b\)](#). From a broad hybrid networked systems viewpoint, our robotic network model can be regarded as special cases of the general modeling paradigms discussed in [Lygeros et al. \(2003\)](#), [Lynch et al. \(2003\)](#), and [Sanfelice et al. \(2007\)](#).

A key feature of the synchronous robotic network model proposed in this chapter is the adoption of proximity graphs from computational geometry as a basis for our communication model. This design choice is justified by the vast wireless networking literature, where this assumption is made. The simplest communication model, in which two robots communicate only if they are

within a fixed communication range, is a common model adopted, for example, in the studies by [Gupta and Kumar \(2000\)](#), [Li \(2003\)](#), [Lloyd et al. \(2005\)](#), and [Santi \(2005\)](#). These works study the proximity graph solutions to various communication optimization problems; this discipline is referred to as *topology control* (cf., Remark 3.8). Although we focus our presentation on the topological aspect of the communication service, more realistic communication models would include randomness, packet losses, coding, quantization, and delays (see, e.g., [Toh, 2001](#); [Tse and Viswanath, 2005](#)).

Next, we review some literature on emergent and self-organized swarm- ing behaviors in biological groups. Interesting dynamical systems arise in biological networks at multiple levels of resolution, all the way from interactions among molecules and cells ([Miller and Bassler, 2001](#)) to the behavioral ecology of animal groups ([Okubo, 1986](#)). Flocks of birds and schools of fish can travel in formation and act as one (see [Parrish et al., 2002](#)), allowing these animals to defend themselves against predators and protect their territories. Wildebeest and other animals exhibit complex collective

behaviors when migrating, such as obstacle avoidance, leader election, and formation-keeping (see [Sinclair, 1977](#); [Gueron and Levin, 1993](#)). Certain foraging behaviors include individual animals partitioning their environment into non-overlapping zones (see [Barlow, 1974](#)). Honey bees ([Seeley and Buhrman, 1999](#)), gorillas ([Stewart and Harcourt, 1994](#)), and whitefaced capuchins ([Boinski and Campbell, 1995](#)) exhibit synchronized group activities such as initiation of motion and change of travel direction. These remarkable dynamic capabilities are achieved apparently without following a group leader; see [Barlow \(1974\)](#), [Okubo \(1986\)](#), [Gueron and Levin \(1993\)](#), [Stewart and Harcourt \(1994\)](#), [Seeley and Buhrman \(1999\)](#), [Boinski and Campbell \(1995\)](#), and [Parrish et al. \(2002\)](#) for specific examples of animal species, and [Conradt and Roper \(2003\)](#), and [Couzin et al. \(2005\)](#) for general studies. A comprehensive exposition of bio-inspired optimization and control methods is presented in [Passino \(2004\)](#).

Regarding distributed motion coordination algorithms, much progress has been made on collective pattern formation ([Suzuki and Yamashita, 1999](#); [Belta and Kumar, 2004](#); [Justh and Krishnaprasad, 2004](#); [Sepulchre et al., 2007](#); [Paley et al., 2007](#); [Yang et al., 2008](#)), flocking ([Olfati-Saber, 2006](#); [Lee and Spong, 2007](#); [Tanner et al., 2007](#); [Moshtagh and Jadbabaie, 2007](#)), motion feasibility of formations ([Tabuada et al., 2005](#)), formation control using rigidity and persistence theory ([Olfati-Saber and Murray, 2002](#); [Baillieul and Suri, 2003](#); [Hendrickx et al., 2007](#); [Krick, 2007](#); [Yu et al., 2009](#)), formation stability ([Tanner et al., 2004](#); [Lafferriere et al., 2005](#); [Kang et al., 2006](#); [Dunbar and Murray, 2006](#); [Smith and Hadaegh, 2007](#); [Zheng et al., 2008](#)), motion camouflage ([Justh and Krishnaprasad, 2006](#)), self-assembly ([Klavins et al., 2006](#)), swarm aggregation ([Gazi and Passino, 2003](#)), gradient climbing ([Ogren et al., 2004](#); [Cortés, 2007](#)), cyclic-pursuit ([Bruckstein et al., 1991](#); [Marshall et al., 2004](#); [Martínez and Bullo, 2006](#); [Smith et al., 2005](#); [Pavone and Frazzoli,](#)

2007), vehicle routing (Sharma et al., 2007), motion planning with collision avoidance (Lumelsky and Harinarayan, 1997; Hu et al., 2007; Pallottino et al., 2007), and cooperative boundary estimation (Bertozzi et al., 2004; Zhang and Leonard, 2005; Clark and Fierro, 2007; Casbeer et al., 2006; Susca et al., 2008). It is also worth mentioning works on network localization, estimation, and tracking (see, e.g., Aspnes et al., 2006; Barooah and Hespanha, 2007; Oh et al., 2007; and the references therein).

Much research has been devoted to distributed task allocation problems. The work in (Gerkey and Mataric, 2004) proposes a taxonomy of task allocation problems. In papers such as (Godwin et al., 2006; Alighanbary and How, 2006; Schumacher et al., 2003; Moore and Passino, 2007; Tang and Özgüner, 2005), advanced heuristic methods are developed, and their effectiveness is demonstrated through analysis, simulation or real world implementation. Distributed auction algorithms are discussed in (Castañón and Wu, 2003;

Moore and Passino, 2007) building on the classic works in (Bertsekas and Castañón, 1991, 1993). A distributed mixed-integer-linear-programming solver is proposed in (Alighanbari and How, 2006). A spatially distributed receding-horizon scheme is proposed in (Frazzoli and Bullo, 2004; Pavone et al., 2007). There has also been prior work on target assignment problems (Beard et al., 2002; Arslan et al., 2007; Zavlanos and Pappas, 2007a; Smith and Bullo, 2009). Target allocation for vehicles with nonholonomic constraints is studied in (Rathinam et al., 2007; Savla et al., 2008, 2009a).

3.6 PROOFS

This section gathers the proofs of the main results presented in the chapter.

3.6.1 Proof of Proposition 3.31

Proof. Let $t \mapsto x(t)$ and $t \mapsto \tilde{x}(t)$ denote the network evolutions starting

from $x_0 \Leftarrow \bigcup_{i \in I} X_0^{[i]}$ under CC and $\text{CC}_{(S, P_r)}$, respectively. From the definition of rescheduling, one can verify that, for all $k \in \mathbb{Z}_{\geq 0}$,

$$\tilde{x}^{[i]}(t) = \tilde{x}^{[i]}(F(k-1)+1), \quad \text{for } t \in \bigcup_{\ell \in [F(k-1)+1, F(k)]} [\ell, \ell+1], \quad (3.6.1)$$

$$x^{[i]}(t - F(k) + k), \quad \text{for } t \in [F(k), F(k) + 1].$$

By the definition of time complexity $\text{TC}(\mathcal{T}, \text{CC}, x_0)$, we have $\mathcal{T}(x(k)) = \text{true}$, for all $k \geq \text{TC}(\mathcal{T}, \text{CC}, x_0)$, and $\mathcal{T}(x(\text{TC}(\mathcal{T}, \text{CC}, x_0) - 1)) = \text{false}$. We rewrite these equalities in terms of the trajectories of $\text{CC}_{(S, P_r)}$. From (3.6.1), we write $x^{[i]}(k) = \tilde{x}^{[i]}(F(k))$, for all $i \in I$ and $k \in \mathbb{Z}_{\geq 0}$. Therefore, we have

$$\mathcal{T}(\tilde{x}(F(k))) = \mathcal{T}(x(k)) = \text{true}, \quad \text{for all } F(k) \geq F(\text{TC}(\mathcal{T}, \text{CC}, x_0)),$$

$$T(\tilde{x}(F(TC(T, CC, x_0) - 1))) = T(x(TC(T, CC, x_0) - 1)) = \text{false},$$

where we have used the rescheduled message-generation function in (3.3.2).

Now, note that by equation (3.6.1), $\tilde{x}^{[i]}(\ell) = \tilde{x}^{[i]}(F(\lfloor \ell/s \rfloor - 1) + 1)$, for all $\ell \in \mathbb{Z}_{\geq 0}$ and all $i \in I$. Therefore, $T(\tilde{x}(F(TC(T, CC, x_0) - 1) + 1)) = T(\tilde{x}(F(TC(T, CC, x_0))))$ and we can rewrite the previous identities as

$$T(\tilde{x}(k)) = \text{true}, \quad \text{for all } k \geq F(TC(T, CC, x_0) - 1) + 1,$$

$$T(\tilde{x}(F(TC(T, CC, x_0) - 1))) = \text{false},$$

which implies $TC(T, CC_{(s, P_r)}, x_0) = F(TC(T, CC, x_0) - 1) + 1 = s TC(T, CC, x_0)$.

As for the mean communication complexity, additivity of C_{rnd} implies

$$\begin{aligned} & C_{\text{rnd}} \circ \mathbf{M}(\ell, x(\ell)) \\ &= C_{\text{rnd}} \circ \mathbf{M}(F(\ell) - s + 1, \tilde{x}(F(\ell) - s + 1)) + \cdots + C_{\text{rnd}} \circ \mathbf{M}(F(\ell), \tilde{x}(F(\ell))), \end{aligned}$$

where we have used $F(\ell) + 1 = F(\ell) + s + 1$.
We conclude the proof by computing

$$\begin{aligned}
& \sum_{\ell=0}^{TC(T, CC(s, P, \cdot), X_0)-1} C_{\text{rnd}} \circ \mathbf{M}(\ell, \tilde{x}(\ell)) = \sum_{\ell=0}^{F(\ell)-1} C_{\text{rnd}} \circ \mathbf{M}(\ell, \tilde{x}(\ell)) \\
& = \sum_{\ell=0}^{TC(T, CC, X_0)-1} C_{\text{rnd}} \circ \mathbf{M}(k, \tilde{x}(k)) \\
& = \sum_{\ell=0}^{TC(T, CC, X_0)-1} C_{\text{rnd}} \circ \mathbf{M}(\ell, x(\ell)).
\end{aligned}$$

□

3.6.2 Proof of Theorem 3.33

Proof. In the following four *STEPS*, we prove the two upper bounds and the two lower bounds.

STEP 1: We start by proving the upper bound in statement (i). We claim that $TC(\text{dir}, \text{agree \& pursue}) \leq 2\pi/(k_{\text{prop}}r(n))$, and we reason by contradiction, that is, we assume that there exists an initial condition which gives rise to an execution with time complexity strictly larger than $2\pi/(k_{\text{prop}}r(n))$. Without loss of generality, assume $\text{dir}^{[n]}(0) = c$. For $\ell \leq 2\pi/(k_{\text{prop}}r(n))$, define

$$k(\ell) = \text{argmin}\{\text{dist}_{cc}(\theta^{[n]}(0), \theta^{[i]}(\ell)) \mid \text{dir}^{[i]}(\ell) = c, i \in \{1, \dots, n\}\}.$$

In other words, agent $k(\ell)$ is the agent moving counterclockwise that has smallest counterclockwise distance from the initial position of agent n . Note that $k(\ell)$ is well-defined since, by hypothesis of contradiction, T_{dir}

is false
for $\ell \leq 2\pi/(k_{\text{prop}}r(n))$. According to the state-transition function of the law $\mathcal{C}_{\text{agree \& pursue}}$ (cf., Section 3.1.3), messages with $\text{dir} = \text{cc}$ can only travel counterclockwise, while messages with $\text{dir} = \text{c}$ can only travel clockwise. Therefore, the position of agent $k(\ell)$ at time ℓ can only belong to the counterclockwise interval from the position of agent $k(0)$ at time 0 to the position of agent n at time 0.

Let us examine how fast the message from agent n travels clockwise. To this end, for $\ell \leq 2\pi/(k_{\text{prop}}r(n))$, define

$$j(\ell) = \text{argmax}\{\text{dist}_c(\theta_{[n]}(0), \theta_{[i]}(\ell)) \mid \text{max-id}_{[i]}(\ell) = n, i \in \{1, \dots, n\}\}.$$

In other words, agent $j(\ell)$ has max-id equal to n , is moving clockwise, and

is the agent furthest from the initial position of agent n in the clockwise direction with these two properties. Initially, $j(0) = n$. Additionally, for $\ell \leq 2\pi/(k_{\text{prop}}r(n))$, we claim that

$$\text{dist}_c(\theta^{j(0)}(\ell), \theta^{j(\ell+1)}(\ell+1)) \geq k_{\text{prop}}r(n).$$

This happens because either (1) there is no agent clockwise-ahead of $\theta^{j(0)}(\ell)$ within clockwise distance $r(n)$, and therefore, the claim is obvious, or (2)

there are such agents. In case (2), let m denote the agent whose clockwise distance to agent $j(\ell)$ is maximal within the set of agents with clockwise distance $r(n)$ from $\theta^{j(0)}(\ell)$.

Then,

$$\begin{aligned} & \text{dist}_c(\theta^{j(0)}(\ell), \theta^{j(\ell+1)}(\ell+1)) \\ &= \text{dist}_c(\theta^{j(0)}(\ell), \theta^{[m]}(\ell+1)) \\ &= \text{dist}_c(\theta^{j(0)}(\ell), \theta^{[m]}(\ell)) + \text{dist}_c(\theta^{[m]}(\ell), \theta^{[m]}(\ell+1)) \\ &\geq \text{dist}_c(\theta^{j(0)}(\ell), \theta^{[m]}(\ell)) + k_{\text{prop}}r(n) - \text{dist}_c(\theta^{j(0)}(\ell), \theta^{[m]}(\ell)) \\ &= k_{\text{prop}}r(n) + (1 - k_{\text{prop}}) \text{dist}_c(\theta^{j(0)}(\ell), \theta^{[m]}(\ell)) \geq k_{\text{prop}}r(n), \end{aligned}$$

where the first inequality follows from the fact that at time ℓ there can be no agent whose clockwise distance to agent m is less than $(r(n) - \text{dist}_c(\theta^{j(0)}(\ell), \theta^{[m]}(\ell)))$.

Therefore, after a number of communication rounds larger than $2\pi/(k_{\text{prop}}r(n))$, the message with $\text{max-id} = n$ has traveled the whole circle in the clockwise direction, and must therefore have reached agent $k(\ell)$. This is a contradiction.

STEP 2: We prove the lower bound in statement (i). If $r(n) > \pi$ for all n , then $1/r(n) < 1/\pi$, and the upper bound is $\text{TC}(\text{dir}, \text{agree} \ \& \ \text{pursue}) = O(1)$. Obviously, the time complexity of any evolution with an initial con-

figuration where $\text{dir}^{[i]}(0) = \text{cc}$ for $i \in \{1, \dots, n-1\}$, $\text{dir}^{[n]}(0) = \text{c}$, and

$E_{G_{\text{disk}(r)}}(\theta^{[1]}(0), \dots, \theta^{[n]}(0))$ is the complete graph, is lower bounded by 1.

Therefore, $\text{TC}(\overline{\text{T}}_{\text{dir}}, \overline{\text{CC}}_{\text{agree \& pursue}}) \in \Omega(1)$. If $r(n) > \pi$ for all n , then we conclude $\text{TC}(\overline{\text{T}}_{\text{dir}}, \overline{\text{CC}}_{\text{agree \& pursue}}) \in \Theta(r(n)^{-1})$.

Assume now that

$r(n) \leq \pi$ for sufficiently large n . Consider an initial configuration where $\text{dir}^{[i]}(0) = \text{cc}$ for $i = 1, \dots, n-1$,

$\text{dir}^{[n]}(0) = \text{c}$, and the agents are placed as depicted in

Figure 3.10. Note that, after each communication

round, agent 1 has moved $k_{\text{prop}}r(n)$ in the counterclockwise direction, while agent n has moved $k_{\text{prop}}r(n)$ in the clockwise direction. These two agents keep moving at full speed toward each other until they become neighbors at a time lower bounded by

$$\frac{2\pi - r(n)}{2k_{\text{prop}}r(n)} > \frac{\pi}{k_{\text{prop}}r(n)} - 1.$$

We conclude that $\text{TC}(\overline{\text{T}}_{\text{dir}}, \overline{\text{CC}}_{\text{agree \& pursue}}) \in \Omega(r(n)^{-1})$.

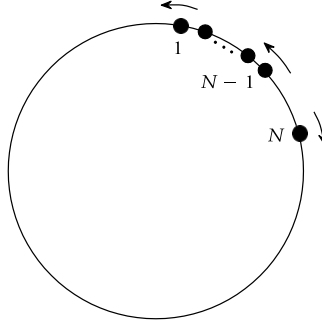


Figure 3.10 Initial condition for the lower bound of $\text{TC}(\text{T}_{\text{dir}}, \text{CC}_{\text{agree \& pursue}})$, with $0 < \text{dist}_c(\theta^{n-1}(0), \theta^n(0)) - r(n) < \varepsilon$ and $\text{dist}_c(\theta^1(0), \theta^{n-1}(0)) \leq r(n) - \varepsilon$, for some fixed $\varepsilon > 0$.

STEP 3: We now prove the upper bound in (ii). We begin by noting that the lower bound on δ implies $r(n)^{-1} \in O(n)$. Therefore, we know that $\text{TC}(\text{T}_{\text{dir}}, \text{CC}_{\text{agree \& pursue}})$ belongs to $O(n)$ and is negligible as compared with the claimed upper bound estimates for $\text{TC}(\text{T}_{\varepsilon\text{-eqdstnc}}, \text{CC}_{\text{agree \& pursue}})$.

In what follows, we therefore assume that dir has been achieved and that,

without loss of generality, all agents are moving clockwise. We now prove a fact regarding connectivity. At time $\ell \in \mathbb{Z}_{\geq 0}$, let $H(\ell)$ be the union of all the empty “circular segments” of length at least $r(n)$, that is, let $H(\ell) = \{x \in S^1 \mid \min_{i \in \{1, \dots, n\}} \text{dist}_c(x, \theta^{(i)}(\ell)) + \min_{j \in \{1, \dots, n\}} \text{dist}_{cc}(x, \theta^{(j)}(\ell)) > r(n)\}$.

$$i \in \{1, \dots, n\}$$

$$j \in \{1, \dots, n\}$$

In other words, $H(\ell)$ does not contain any point between two agents separated by a distance less than $r(n)$, and each connected component of $H(\ell)$ has length at least $r(n)$. Let $n_H(\ell)$ be the number of connected components of $H(\ell)$; if $H(\ell)$ is empty, then we take the convention that $n_H(\ell) = 0$.

Clearly, $n_H \leq n$. We claim that if $n_H(\ell) > 0$, then $n_H(\ell + \tau)$ is non-increasing. Let $d(\ell) < r(n)$ be the distance between any two consecutive agents at time ℓ . Because both agents move in the same direction, a simple calculation shows that

$$\begin{aligned}
d(\ell + 1) &\leq d(\ell) + k_{\text{prop}}(r - d(\ell)) = (1 - k_{\text{prop}})d(\ell) + k_{\text{prop}}r(n) \\
&< (1 - k_{\text{prop}})r + k_{\text{prop}}r(n) = r(n).
\end{aligned}$$

This means that the two agents remain within distance $r(n)$ and, therefore, connected at the following time instant. Because the number of connected components $\text{disk}(r)(\theta^{[1]}, \dots, \theta^{[n]})$ does not increase, it follows that the of E_G

number of connected components of H cannot increase.

Next, we claim that if $n_H(\ell) > 0$, then there exists $\tau >$

ℓ such that $n_H(\tau) < n_H(\ell)$. By

contradiction, assume that $n_H(\ell) = n_H(\tau)$ for all $\tau \geq \ell$.

Without loss of generality, let $\{1, \dots, m\}$ be a set of agents with the properties that

$\text{dist}_{\text{cc}}^{[i]}(\ell), \theta^{[i+1]}(\ell) \leq r(n)$, for $i \in \{1, \dots, m\}$, that $\theta^{[1]}(\ell)$ and $\theta^{[m]}(\ell)$

θ

17

0

belong to the boundary of $H(\ell)$, and that there is no other set with the same properties and more agents. (Note that this implies that the agents $1, \dots, m$ are in counterclockwise order.) One can show that, for $\tau \geq \ell$,

$$\theta^{[1]}(\tau + 1) = \theta^{[1]}(\tau) - k_{\text{prop}} r(n),$$

$$\theta^{[i]}(\tau + 1) = \theta^{[i]}(\tau) - k_{\text{prop}} \text{dist}_c(\theta^{[i]}(\tau), \theta^{[i-1]}(\tau)), \text{ for } i$$

$\in \{2, \dots, m\}$. If we consider the inter-agent distances

$$d(\tau) = \frac{1}{\Sigma} \left(\text{dist}(\theta^{[1]}(\tau), \theta^{[2]}(\tau)), \dots, \text{dist}(\theta^{[m-1]}(\tau), \theta^{[m]}(\tau)) \right) \in \mathbb{R}_{>0}^{m-1},$$

Σ

then the previous equations can be rewritten as

$$d(\tau + 1) = \text{Trid}_{m-1}(k_{\text{prop}}, 1 - k_{\text{prop}}, 0) d(\tau) + r(n) k_{\text{prop}} \mathbf{e}_1,$$

where the linear map $(a, b, c) \mapsto \text{Trid}_{m-1}(a, b, c)$

$\mathbb{R}^{(m-1) \times (m-1)}$ is defined in Section 1.6.4.

This is a discrete-time affine time-invariant dynamical

system with unique equilibrium point $r(n) \mathbf{1}_{m-1}$. By construction, the initial

condition of this system satisfies $\|d(0) - r(n) \mathbf{1}_{m-1}\|_2 \leq r(n)$

$m - 1$. By Theorem 1.79(ii) in Section 1.6.4, for $\eta_1 \in]0, 1[$, the solution $\tau \mapsto d(\tau)$ to this

system reaches a ball of radius η_1 centered at the equilibrium point in time

$O(m \log m + \log \eta_1^{-1})$. (Here we have used the fact that the initial condition

of this system is bounded.) In turn, this implies that $\tau \mapsto \sum_{i=1}^m d_i(\tau)$ is larger than $(m - 1)(r(n) - \eta_1)$ in time $O(m \log m + \log \eta_1^{-1})$. We are now

ready to find the contradiction, and show that $n_H(\tau)$

cannot remain equal to $n_H(\ell)$ for all time τ . After time

$O(m \log m + \log \eta_1^{-1}) = O(n \log n + \log \eta_1^{-1})$, we have

$$2\pi \geq n_H(\ell) r(n) + \sum_{j=1}^{n_H(\ell)} (r(n) - \eta_1)(m_j - 1)$$

$$= n_H(\ell)r(n) + (n - n_H(\ell))(r(n) - \eta_1) = n_H(\ell)\eta_1 + n(r(n) - \eta_1).$$

Here, $m_1, \dots, m_{n_H(\ell)}$ are the numbers of agents in each isolated group, and each connected component of $H(\ell)$ has length at least $r(n)$. Now, take $\eta_1 = (nr(n) - 2\pi)n^{-1} = \delta(n)n^{-1}$, and the contradiction follows from

$$2\pi \geq n_H(\ell)\eta_1 + nr(n) - n\eta_1$$

$$= n_H(\ell)\eta_1 + nr(n) + 2\pi - nr(n) = n_H(\ell)\eta_1 + 2\pi.$$

In summary, this shows that the number of connected components of $H(\ell)$ decreases by one in time $O(n \log n + \log \eta_1^{-1}) = O(n \log n + \log(n\delta(n)^{-1}))$. Note that δ being lower bounded implies that $n\delta(n)^{-1} = O(n)$ and, therefore, $O(n \log n + \log(n\delta(n)^{-1})) = O(n \log n)$. Iterating this argument n times, in time $O(n^2 \log n)$ the set H will become empty. At that time, the resulting network will obey the discrete-time linear time-invariant dynamical system

$$d(\tau + 1) = \text{Circ}_n(k_{\text{prop}}, 1 - k_{\text{prop}}, 0) d(\tau), \quad (3.6.2)$$

where the linear map $(a, b, c) \mapsto \text{Circ}_n(a, b, c) \in \mathbb{R}^{n \times n}$ is defined in Section

1.6.4 and where $d : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{R}^n$ is defined by

$$d(\tau) = (\text{dist}_{\text{cc}}(\theta^{[1]}(\tau), \theta^{[2]}(\tau)), \dots, \text{dist}_{\text{cc}}(\theta^{[n]}(\tau), \theta^{[n+1]}(\tau)))^\top,$$

with the convention $\theta^{[n+1]} = \theta^{[1]}$. By Theorem 1.79(iii) in Section

1.6.4, the 2-norm inequality 2-norm satisfies the contraction $\|d(\tau) - d_*(\tau)\|_2 \leq \varepsilon \|d(0) - d_*(0)\|_2$, for $d_* = \frac{1}{n} \mathbf{1}_n$. We convert this inequality on

2-norms into an appropriate inequality on ∞ -norms as follows. For $d \in \mathbb{R}^n$, let $\|d\|_\infty = \max_{i \in \{1, \dots, n\}} |d^{(i)}|$. For $\eta_2 \in]0, 1[$ and for τ of

order $n^2 \log \eta_2^{-1}$, $i = 1, \dots, n$ *

$$\begin{aligned} \|d(\tau) - d_*\|_\infty &\leq \|d(\tau) - d_*\|_2 \leq \eta_2 \|d(0) - d_*\|_2 \\ &\leq \eta_2 \sqrt{n} \|d(0) - d_*\|_\infty \leq \eta_2 2\pi / n. \end{aligned}$$

This means that the desired configuration is achieved for $\eta_2 2\pi / n = \varepsilon$, that is, in time $O(n^2 \log \eta_2^{-1}) = O(n^2 \log(n\varepsilon^{-1}))$. In summary, the equidistance task is achieved in time $O(n^2 \log(n\varepsilon^{-1}))$.

STEP 4: Finally, we prove the lower bound in (ii). As we reasoned before, $\text{TC}(\text{T}_{\text{dir}}, \text{CC}_{\text{agree \& pursue}})$ is negligible as compared with the claimed lower bound estimate for $\text{TC}(\text{T}_{\varepsilon\text{-eqdstnc}}, \text{CC}_{\text{agree \& pursue}})$ and, therefore, we assume that T_{dir} has been achieved. We consider an initial configuration

with the properties that: (i) agents are counterclockwise-ordered according to their unique identifier; (ii) the set $H(0)$ is empty; and (iii) the inter-agent

distances $d(0) = (\text{dist}_{\text{cc}}(\theta^{[1]}(0), \theta^{[2]}(0)), \dots, \text{dist}_{\text{cc}}(\theta^{[n]}(0), \theta^{[1]}(0)))^\top$ are

$$d(0) = \frac{1}{n} \left(\frac{2\pi}{n} + \frac{\pi - \varepsilon'}{n} \langle v_n + v_n \rangle \right),$$

where $\varepsilon' \in]\pi, 0[$ and where v_n is the eigenvector of $\text{Circ}_n(k_{\text{prop}}, 1)$ corresponding to the eigenvalue $1 - k + k \cos k$

$= k_{\text{prop}}, 0)$

$$= \frac{1}{n} \sum_{i=1}^n \left(\frac{1}{n} \sum_{j=1}^n \cos((i-j)2\pi/n) \right) = \frac{1}{n^2} \sum_{i,j=1}^n \cos((i-j)2\pi/n)$$

(see Section 1.6.4). Straightforward calculations show the equality $\sum_{i=1}^n \cos((i-j)2\pi/n) = \cos((j-1)2\pi/n) \sum_{i=1}^n \cos((i-1)2\pi/n)$ and that $\|v_n + v_n\|_2 = \sqrt{2n}$. In turn,

this implies that $d(0) \in \mathbb{R}^n$, $\|d(0)\|_2 = \sqrt{2n}$ and that $\|d(0) - \frac{1}{\sqrt{2n}} \mathbf{1}_n\|_2 \in O(1/\sqrt{n})$. Take

$\eta_3 \in]0, 1[$. The argument described in the proof of Theorem 1.79(iii) leads

to the following statement: the 2-norm of the difference between ℓ and the desired configuration $\frac{1}{\sqrt{2n}} \mathbf{1}_n$ decreases by a factor $\eta_3^{d(\ell)}$ in time of order $\frac{1}{\eta_3^{d(\ell)}}$

$\frac{1}{\eta_3} \log \eta_3^{-1}$. Given an initial error of order $O(1/\sqrt{n})$ and a final desired error of order ε , we set $\eta_3^{-1} = \varepsilon \sqrt{n}$ and obtain the desired result that it takes time

of order $n^2 \log(n\varepsilon)^{-1}$ to reduce the 2-norm error and, therefore, the

norm error to size ε . This concludes the proof. $\square$

3.6.3 Proof of Theorem 3.34

Proof. Note that the number of edges in circle is at most $O(n^2)$, as it is possible that all robots are within distance $r(n)$ of each other. The upper bounds in (i) and (ii) then follow from inequality (3.3.1) and Theorem 3.33. To prove the lower bounds, we follow the steps and notation in the proof of Theorem 3.33. Regarding the lower bounds in (i), we examine the evolution of the initial configuration depicted in Figure 3.10. From STEP 2: in the proof of Theorem 3.33, recall that the time it takes agent 1 to receive the message with $\text{max-id} = n$ is lower bounded by $\pi/(k_{\text{prop}}r(n))$. Our proof strategy is to lower bound the number of edges in the graph until this event happens. Note that, at initial time, there are $(n - 1)^2$ edges in the communication graph of the network and, therefore, $(n - 1)^2$ messages get transmitted. At the next communication round, agent 1 has moved $k_{\text{prop}}r(n)$ counterclockwise and, therefore, the number of edges is lower bounded by $(n - 2)^2$. Iterating this reasoning, we see that after $i < \pi/(k_{\text{prop}}r(n))$ communication rounds, the number of edges is lower bounded by $(n - i)^2$. Now, if $\delta(n) > \pi(1/k_{\text{prop}} - 2)$, then $n > \pi/k_{\text{prop}}r(n)$, and therefore, the total communication complexity is lower bounded by

$$\sum_{i=1}^{\frac{\pi}{k_{\text{prop}}r(n)}} (n - i)^2 \in \Omega(n^2r(n)^{-1}).$$

On the other hand, if $\delta(n) < \pi(1/k_{\text{prop}} - 2)$, then $n < \pi/k_{\text{prop}}r(n)$, and after

n time steps, we lower bound the number of edges in the communication graph by the number of edges in a chain of length n , that is, $n - 1$. Therefore, the total communication complexity is lower bounded by

$$\sum_{i=1}^n (n - i)^2 + (n - 1) \cdot \frac{\pi}{k_{\text{prop}}r(n)} - n \in \Omega(n^3 + nr(n)^{-1}).$$

The two lower bounds match when $\delta(n) = \pi(1/k_{\text{prop}} - 2)$.

Regarding the lower bound in (ii), we consider first the case

when $n_H(0) = 0$. In this case, the network obeys the discrete-time linear time-invariant dynamical system (3.6.2). Consider the initial condition $d(0)$ that we adopted for *STEP 4*:. We know it takes time of order $n^2 \log(n\varepsilon)^{-1}$ for the appropriate contraction property to hold. At $d(0)$, the maximal inter-agent distance is $(4\pi - \varepsilon')/n$ and it decreases during the evolution. Because each robot can communicate with any other robot within a distance $r(n)$, the number of agents within communication range of a given agent is of order $r(n)n/(4\pi - \varepsilon')$, that is, of order $\delta(n)$. From here, we deduce that the total communication complexity belongs to $\Omega(n^3\delta(n) \log(n\varepsilon)^{-1})$.

□

Chapter Four

Connectivity maintenance and rendezvous

The aims of this chapter are twofold. First, we introduce the rendezvous problem and analyze various coordination algorithms that achieve it, providing upper and lower bounds on their time complexity. Second, we introduce the problem of maintaining connectivity among a group of mobile robots and use geometric approaches to preserve this topological property of the network.

Loosely speaking, the *rendezvous objective* is to achieve agreement over the physical location of as many robots as possible, that is, to steer the robots to a common location. This objective is to be achieved with the limited information flow described in the model of the network. Typically, it will be impossible to solve the rendezvous problem for all robots if the robots are placed in such a way that they do not form a connected communication graph. Therefore, it is reasonable to assume that the network is connected at initial time, and that a good property of any rendezvous algorithm is that of maintaining some form of connectivity among robots. This discussion motivates the *connectivity maintenance problem*. Once a model for when two robots can acquire each other's relative position is adopted, this problem is of particular relevance, as the inter-robot topology depends on the physical states of the robots. Our exposition here is mainly based on [Ando et al. \(1999\)](#), [Cortés et al. \(2006\)](#), and [Ganguli et al. \(2009\)](#).

The chapter is organized as follows. In the first section, we formally introduce the two coordination problems. In the second section, we define various connectivity constraint sets to limit the motion of robots in order to maintain network connectivity. These notions of constraint sets allows us to study in the next section various rendezvous algorithms with connectivity maintenance properties. We study numerous variations of the circumcenter algorithm for the rendezvous objective and we characterize its complexity. Additionally, we introduce the perimeter-minimizing algorithm for nonconvex environments. The fourth section presents various simulations of the proposed motion-coordination algorithms. We end the chapter with three sections on, respectively, bibliographic notes, proofs of the results

presented in the chapter, and exercises. Our technical treatment is based on the LaSalle Invariance Principle, on linear distributed algorithms, and on geometric tools such as proximity graphs and robust visibility.

4.1 PROBLEM STATEMENT

We begin this section by reviewing the classes of networks and the types of problems that will be considered in the chapter.

4.1.1 Networks with discrete-time motion

In the course of the chapter, we will consider the robotic networks S_{disk} , S_{LD} , and $S_{\infty\text{-disk}}$, and the relative-sensing networks $S_{\text{disk}}^{\text{rs}}$ and $S_{\text{vis-disk}}^{\text{rs}}$ presented in Example 3.4 and in Section 3.2.2.

For the robotic networks S_{disk} , S_{LD} , and $S_{\infty\text{-disk}}$, we will, however, assume that the robots move in discrete time, that is, we adopt the discrete-time motion model

$$p^{[i]}(\ell + 1) = p^{[i]}(\ell) + u^{[i]}(\ell), \quad i \in \{1, \dots, n\}. \quad (4.1.1)$$

Similarly, for the relative-sensing networks $S_{\text{disk}}^{\text{rs}}$ and $S_{\text{vis-disk}}^{\text{rs}}$, we adopt the

discrete-time motion model

$$p_{\text{fixed}}^{[i]}(\ell + 1) = p_{\text{fixed}}^{[i]}(\ell) + R_{\text{fixed}}^{[i]} u_{\text{fixed}}^{[i]}(\ell), \quad i \in \{1, \dots, n\}. \quad (4.1.2)$$

As an aside, if we express the previous equation with respect to frame i at time t , then equation (4.1.2) reads

$$p_{(\text{frame } i \text{ at time } t)}^{[i]}(\ell + 1) = u_{(\text{frame } i \text{ at time } t)}^{[i]}(\ell), \quad i \in \{1, \dots, n\}.$$

We present the treatment in discrete time for simplicity. It is easy to show that any control law for the discrete-time motion model can be implemented in the continuous-time networks. In what follows, we begin our discussion by assuming no bound on the control magnitude and we later introduce an upper bound denoted by $u_{\max}$.

4.1.2 The rendezvous task

Next, we discuss the rendezvous problem. There are different ways of formulating this objective in terms of task maps. Let $S = (\{1, \dots, n\}, R, E_{\text{cmm}})$ be a uniform robotic network. The *(exact) rendezvous task* $T_{\text{rndzvs}} : X^n \rightarrow$

$\{\text{true}, \text{false}\}$ for S is the coordination task defined by

$$\begin{aligned} T_{\text{rndzvs}}(x^{[1]}, \dots, x^{[n]}) \\ = \begin{cases} \text{true}, & \text{if } x^{[i]} = x^{[j]}, \text{ for all } (i, j) \in E_{\text{cmm}}(x^{[1]}, \dots, x^{[n]}), \\ \text{false}, & \text{otherwise.} \end{cases} \end{aligned}$$

Next, assume that, for the same network $G = (V, E_{\text{cmm}})$, the robots' physical state space is $X = \mathbb{R}^d$. It is convenient to review some basic notation consistent with what we adopted in Chapter 2. We let $\{p^{[1]}, \dots, p^{[n]}\}$ denote the set of agents' location in $X = \mathbb{R}^d$ and we let P be an array of n points in $\mathbb{R}^d$. Furthermore, we let avg denote the average of a finite point set in $\mathbb{R}^d$, that is,

$$\text{avg}(\{q_1, \dots, q_k\}) = \frac{1}{k} (q_1 + \dots + q_k).$$

For $\varepsilon \in \mathbb{R}_{>0}$, the ε -rendezvous task $T_{\varepsilon\text{-rndzvs}} : (\mathbb{R}^d)^n \rightarrow \{\text{true}, \text{false}\}$ for S is defined as follows: $T_{\varepsilon\text{-rndzvs}}$ is true at P if and only if each robot position $p^{[i]}$, for $i \in \{1, \dots, n\}$, is at distance less than ε from the average position of its E_{cmm} -neighbors. Formally,

$$\begin{aligned} T_{\varepsilon\text{-rndzvs}}(P) = \text{true} \\ \Leftrightarrow \sum_{i \in \{1, \dots, n\}} \|p^{[i]} - \text{avg}(\{p^{[j]} \mid (i, j) \in E_{\text{cmm}}(P)\})\|_2 < \varepsilon, \end{aligned}$$

4.1.3 The connectivity maintenance problem

Assume that the communication graph, computed as a function of the robot positions, is connected: How should the robots move in such a way that their communication graph is again connected? Clearly, the problem depends upon: (1) how the robots move; and (2) what proximity graph describes the communication graph or, in the case of relative-sensing networks, what sensor model is

available on each robot.

The key idea is to restrict the allowable motion of each agent. Different motion constraint sets correspond to different communication or sensing graphs. We have three objectives in doing so. First, we aim to achieve this objective only based on local measurements or 1-hop communication, that is, without introducing processor states explicitly dedicated to this task. Second, the constraint sets should depend continuously on the position of the robots. Third, we have the somehow informal objective to design the constraint sets as “large” as possible so as to minimally constrain the motion of the robots.

4.2 CONNECTIVITY MAINTENANCE ALGORITHMS

In this section, we present some algorithms that might be used by a robotic network to maintain communication connectivity. The results presented in this section start with the original idea introduced by [Ando et al. \(1999\)](#) for first-order robots communicating along the edges of a disk graph, that is, for the network described in Example 3.4. This idea is then generalized to a number of useful settings. The properties of proximity graphs presented in Section 2.2 play a key role in formulating and solving the connectivity problem.

4.2.1 Enforcing range-limited links

First, we aim to constrain the motion of two first-order agents in order to maintain a communication link between them. We assume that the communication takes place over the disk graph $\text{disk}(r)$ with communication range $r > 0$.

Loosely stated, the *pairwise connectivity maintenance problem* is as follows: given two neighbors in the proximity graph $\text{disk}(r)$, find a rich set of control inputs for both agents with the property that, after moving, both agents are again within distance r . We provide a solution to this problem as follows.

Definition 4.1 (Pairwise connectivity constraint set). Consider two agents i and j at positions $p^{[i]} \in \mathbb{R}^d$ and $p^{[j]} \in \mathbb{R}^d$ such that $\|p^{[i]} - p^{[j]}\|_2 \leq r$. The *connectivity constraint set* of agent i with respect to agent j is

$$\mathcal{X}_{\text{disk}}(p^{[i]}, p^{[j]}) = \{x \in \mathbb{R}^d \mid \|x - p^{[j]}\|_2 \leq r\}.$$

Note that both robots, i and j , can independently compute their respective connectivity constraint sets. The proof of the following result is straight-forward.

Lemma 4.2 (Maintaining pairwise connectivity). *Assume that the distance between agents $p^{[i]}$ and $p^{[j]}$ is no more than r , at some time ℓ . If the control $u^{[i]}(\ell)$ takes value in*

$$u^{[i]}(\ell) \in X^{\text{disk}} \left(p^{[i]}(\ell), p^{[j]}(\ell) - p^{[i]}(\ell) = B \frac{p^{[j]}(\ell) - p^{[i]}(\ell)}{\|p^{[j]}(\ell) - p^{[i]}(\ell)\|}, \frac{r}{2} \right),$$

and, similarly, $u^{[j]}(\ell) \in X^{\text{disk}} \left(p^{[j]}(\ell), p^{[i]}(\ell) - p^{[j]}(\ell) = B \frac{p^{[i]}(\ell) - p^{[j]}(\ell)}{\|p^{[i]}(\ell) - p^{[j]}(\ell)\|}, \frac{r}{2} \right)$, then, according to the discrete-time motion model (4.1.1):

- (i) the positions of both agents at time $\ell + 1$ are inside the connectivity constraint set $X_{\text{disk}}(p^{[i]}(\ell), p^{[j]}(\ell))$; and
- (ii) the distance between the agents' positions at time $\ell + 1$ is no more than r .

We illustrate these pairwise connectivity maintenance concepts in Figure 4.1.

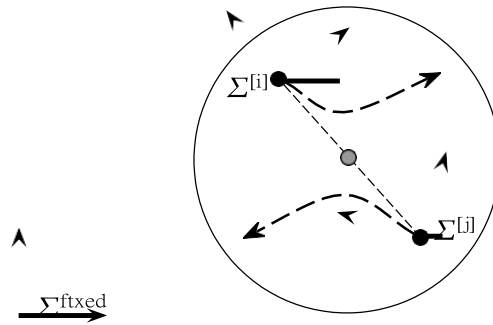


Figure 4.1 An illustration of the connectivity maintenance constraint. Starting from positions $p^{[i]}$ and $p^{[j]}$, the robots are restricted to moving inside the disk centered

$$\text{at } X_{\text{disk}}(p^{[i]}, p^{[j]}) = \frac{1}{2} (p^{[i]} + p^{[j]}) \text{ with radius } r.$$

Remark 4.3 (Constraints for relative-sensing networks). Let us consider a relative-sensing network with a disk sensor of radius r (see Example 3.15). Recall the following facts about this model. First, agent i measures the position of robot j in its frame $\Sigma^{[i]}$, that is, robot i measures $p_i^{[j]}$. Second, $p_i^{[i]} = 0_d$. Third, if $W \subset \mathbb{R}^d$, then W_i denotes its expression in the frame $\Sigma^{[i]}$. Combining these notions and assuming that the inter-agent distance is no more than r , the pairwise connectivity constraint set in Definition 4.1 satisfies

$$X_{\text{disk}}(p^{[i]}, p^{[j]}) = X_{\text{disk}}(0_d, p_i^{[j]}) = B\left(\frac{i}{2}, \frac{r}{2}\right).$$

4.2.2 Enforcing network connectivity

Here, we focus on how to constrain the mobility of multiple agents in order to maintain connectivity for the entire network that they form. We again consider the case of first-order agents moving according to the discrete-time equation (4.1.1) and communicating over $G_{\text{disk}}(r)$.

Loosely stated, the *network connectivity maintenance problem* is as follows: Given n agents at positions $\mathbf{P}(\ell) = \{p^{[1]}(\ell), \dots, p^{[n]}(\ell)\}$ in which they form a connected r -disk graph $\text{disk}(r)$, the objective is to find a rich set

of control inputs for all agents with the property that, at time $\ell + 1$, the agents' new positions $(\ell + 1)$ again form a connected r -disk graph $\text{disk}(r)$. We provide a simple, but potentially conservative, solution to this problem as follows.

Definition 4.4 (Connectivity constraint set for groups of agents). Consider a group of agents at positions $\mathbf{P} = \{p^{[1]}, \dots, p^{[n]}\} \subset \mathbb{R}^d$. The *connectivity constraint set* of agent i with respect to $\mathbf{P}$ is

$$\mathbf{X}_{\text{disk}}(p^{[i]}, \mathbf{P}) = \{x \in \mathbf{X}_{\text{disk}}(p^{[i]}, q) \mid q \in \mathbf{P} \setminus \{p^{[i]}\} \text{ s.t. } \|q - p^{[i]}\|_2 \leq r\}.$$

In other words, if $q_1, \dots, q_l$ are agents' positions whose distance from $p^{[i]}$ is no more than r , then the connectivity constraint set for agent i is the intersection of the constraint sets $\mathbf{X}_{\text{disk}}(p^{[i]}, q_k)$ for $k \in \{1, \dots, l\}$ (see Figure 4.2).

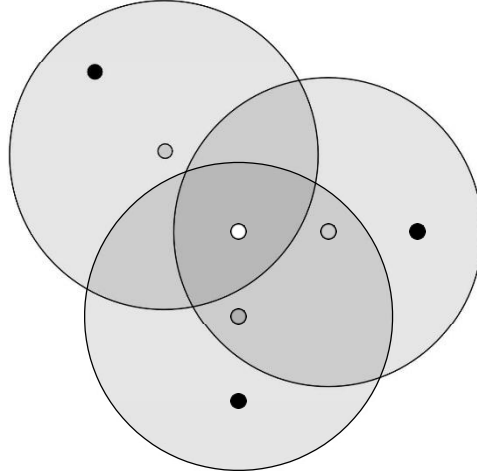


Figure 4.2 An illustration of network connectivity maintenance. The connectivity $\mathbf{X}_{\text{disk}}$ -constraint set of the white-colored agent is the intersection of the individual constraint sets determined by its neighbors.

The following result is a consequence of Lemma 4.2.

Lemma 4.5 (Maintaining network connectivity). *Consider a group of agents at positions $P(\ell) = \{p^{[1]}(\ell), \dots, p^{[n]}(\ell)\} \subset \mathbb{R}^d$ at time ℓ . If each*

agent's control $u^{[i]}(\ell)$ takes value in

$$u^{[i]}(\ell) \in X_{\text{disk}}(p^{[i]}(\ell), P(\ell) - p^{[i]}(\ell)), \quad i \in \{1, \dots, n\},$$

then, according to the discrete-time motion model (4.1.1):

- (i) each agent remains in its connectivity constraint set, that is, $p^{[i]}(\ell + 1) \in X_{\text{disk}}(p^{[i]}(\ell), P(\ell))$;
- (ii) each edge of $G_{\text{disk}}(r)$ at $P(\ell)$ is maintained after the motion step, that is, if $\|p^{[i]}(\ell) - p^{[j]}(\ell)\|_2 \leq r$, then also $\|p^{[i]}(\ell + 1) - p^{[j]}(\ell + 1)\|_2 \leq r$;
- (iii) if $G_{\text{disk}}(r)$ at time ℓ is connected, then $G_{\text{disk}}(r)$ at time $\ell + 1$ is connected; and
- (iv) the number of connected components of the graph $G_{\text{disk}}(r)$ at time $\ell + 1$ is equal to or smaller than the number of connected components of the graph $G_{\text{disk}}(r)$ at time ℓ .

Remark 4.6 (Constraints for relative-sensing networks: cont'd). Following up on Remark 4.3, the connectivity constraint set in Definition 4.4, written in the frame $\Sigma^{[i]}$, is

$$\begin{aligned} X_{\text{disk}}(0_d, \{p_i, \dots, p_j\}) \\ = \left\{ p^{[i]} \in \mathbb{R}^d \mid \exists j \in \{1, \dots, n\} \text{ such that } \|p^{[j]} - p^{[i]}\|_2 \leq r \right\}. \end{aligned}$$

Next, we relax the constraints in Definition 4.4 to provide the network nodes with larger, and therefore less conservative, motion-constraint sets. Recall from Section 2.2 the relative neighborhood graph G_{RN} , the Gabriel graph G_{G} , and the r -limited Delaunay graph $G_{\text{LD}}(r)$. These proximity graphs are illustrated in Figure 2.8. From Theorem 2.8 and Proposition 2.9, respectively, recall that the proximity graphs $G_{\text{RN}} \cap G_{\text{disk}}(r)$, $G_{\text{G}} \cap G_{\text{disk}}(r)$, and $G_{\text{LD}}(r)$ have the following properties:

- (i) they have the same connected components as $G_{\text{disk}}(r)$, that is, for all point sets $P \subset \mathbb{R}^d$, all graphs have the same number of connected components consisting of the same vertices; and

(ii) they are spatially distributed over $G_{\text{disk}}(r)$.

These mathematical facts have two implications. First, to maintain or decrease the number of connected components of a disk graph, it is sufficient to maintain or decrease the number of connected components of any of the three proximity graphs $RN_{\text{disk}}(r)$, $G_{\text{disk}}(r)$, and $LD(r)$. Because each of these graphs is more sparse than the disk graph, that is, they are subgraphs of $G_{\text{disk}}(r)$, fewer connectivity constraints need to be imposed.

Second, because these proximity graphs are spatially distributed over the disk graph, it is possible for each agent to determine which of its neighbors in $G_{\text{disk}}(r)$ are also its neighbors in these subgraphs. We formalize this discussion as follows.

Definition 4.7 (G-connectivity constraint set). Let G be a proximity graph that is spatially distributed over $G_{\text{disk}}(r)$ and that has the same connected components as $G_{\text{disk}}(r)$. Consider a group of agents at positions $\mathbf{P} = \{p^{[1]}, \dots, p^{[n]}\} \subset \mathbb{R}^d$. The *G-connectivity constraint set* of agent i with respect to $\mathbf{P}$ is

$$\begin{aligned} X_{\text{disk},G}(p^{[i]}, \mathbf{P}) &= \{x \in X_{\text{disk}}(p^{[i]}) \mid (x, q) \mid q \in \mathbf{P} \text{ s.t. } (q, p^{[i]}) \text{ is an edge of } G(\mathbf{P})\}. \end{aligned}$$

Lemma 4.8 (Maintaining connectivity of sparser networks). Let G be a proximity graph that is spatially distributed over $G_{\text{disk}}(r)$ and that has the same connected components as $G_{\text{disk}}(r)$. Consider a group of agents at positions $\mathbf{P}(\ell) = \{p^{[1]}(\ell), \dots, p^{[n]}(\ell)\} \subset \mathbb{R}^d$ at time ℓ . If each agent's control

$u^{[i]}(\ell)$ takes value in

$$u^{[i]}(\ell) \in X_{\text{disk},G}(p^{[i]}(\ell), \mathbf{P}(\ell)) - p^{[i]}(\ell), \quad i \in \{1, \dots, n\},$$

then, according to the discrete-time motion model (4.1.1):

- (i) each agent remains in its G-connectivity constraint set;
- (ii) two agents that are in the same connected component of G remain at the same connected component after the motion step; and
- (iii) the number of connected components of the graph at $(\ell + 1)$ is equal to or smaller than the number of connected components of the graph G at $\mathbf{P}(\ell)$.

The reader is asked to provide a proof of this result in Exercise E4.1.

4.2.3 Enforcing range-limited line-of-sight links and network connectivity

Here, we consider the connectivity maintenance problem for a group of agents with range-limited line-of-sight communication, as described in Example 3.6. It is convenient to treat directly and only the case of a compact allowable nonconvex environment Q $\subset \mathbb{R}^2$ contracted into Q_δ

based on $\bar{Q}^q \text{dist}(q, \partial Q) \geq \delta$ for a small positive δ . We present a solution designing constraint sets that guarantee that every edge of the range-limited visibility graph $G_{\text{vis-disk}, Q_\delta}$ is preserved.

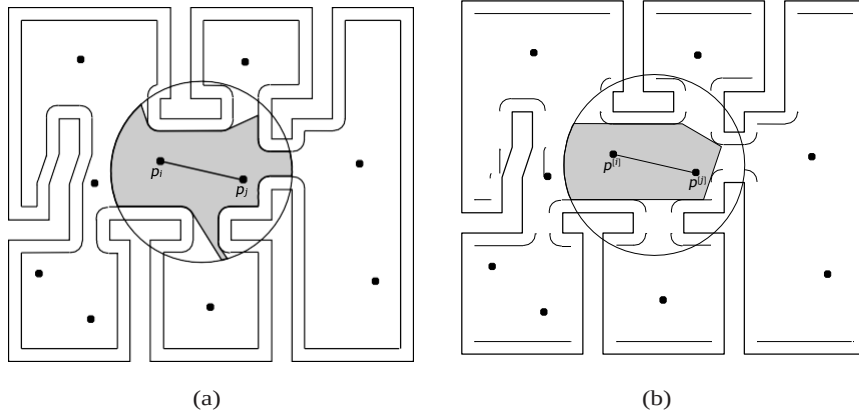


Figure 4.3 Image (a) shows the set $\text{Vi}_{\text{disk}}(p^i; Q_\delta) \cap B(\frac{p^i + p^j}{2}, r)$. Image (b) illustrates the execution of the iterated truncation algorithm. Robots i and j are constrained to remain inside the shaded region in (b), which is a convex subset of Q_δ and of the closed ball with center $\frac{p^i + p^j}{2}$ and radius r .

We begin with a useful observation and a corresponding geometric algorithm. Assume that, at time ℓ , robot j is inside the range-limited visibility set from $p^{[i]}$ in Q_δ , that is, with the notation of Section 2.1.2,

$$p^{[j]}(\ell) \in \text{Vi}_{\text{disk}}(p^{[i]}(\ell); Q_\delta) = \text{Vi}(p^{[i]}(\ell); Q_\delta) \cap B(p^{[i]}(\ell), r).$$

This property holds also at time $\ell + 1$ if $\|p^{[i]}(\ell + 1) - p^{[j]}(\ell + 1)\|_2 \leq r$ and $[p^{[i]}(\ell + 1), p^{[j]}(\ell + 1)] \subset Q_\delta$. A sufficient condition is therefore that

$$p^{[i]}(\ell + 1), p^{[j]}(\ell + 1) \in X,$$

for some convex subset X of $\text{Vi}(p^{[i]}(\ell), p^{[j]}(\ell)) \cap B(\frac{p^{[i]}(\ell) + p^{[j]}(\ell)}{2}, r)$. Intuitively speaking, X plays the role of X -constraint set for the proximity graph. The following geometric algorithm, given the positions $p^{[i]}$ and $p^{[j]}$ in an environment Q_δ , computes precisely one such convex subset:

function **iterated truncation**($p^{[i]}, p^{[j]}; Q_\delta$)
 % Executed by robot i at position $p^{[i]}$ assuming that robot j is at position
 $p^{[j]}$ within range-limited line of sight of $p^{[i]}$

- 1: $\mathbf{X}_{\text{temp}} := \text{Vi}_{\text{disk}}(p^{[i]}; Q_\delta) \cap \underline{B}_2(p^{[i]} + p^{[j]}, \frac{\Sigma}{2})$
- 2: while $\partial\mathbf{X}_{\text{temp}}$ contains a concavity do
- 3: $v :=$ a strictly concave point of $\partial\mathbf{X}_{\text{temp}}$ closest to $[p^{[i]}, p^{[j]}]$ 4:
- $\mathbf{X}_{\text{temp}} := \mathbf{X}_{\text{temp}} \cap H_{Q_\delta}(v)$
- 5: return $\mathbf{X}_{\text{temp}}$

Note: in step 3: multiple points belonging to distinct concavities may satisfy the required property. If so, v may be chosen as any of them.

Figure 4.3 illustrates an example convex constraint set computed by the

iterated truncation algorithm. Figure 4.4 illustrates the step-by-step execution required to generate Figure 4.3(b).

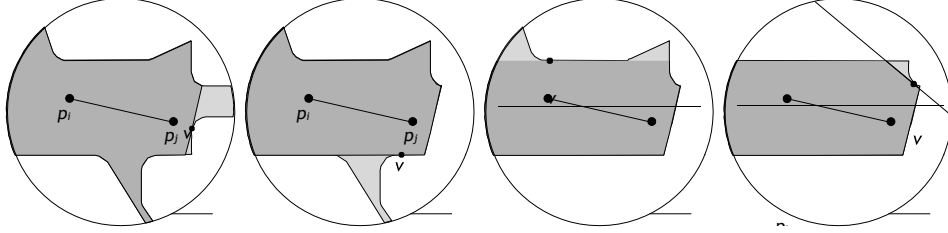


Figure 4.4 From left to right, a sample run of the iterated truncation algorithm. The set $X_{\text{temp}} := V_{\text{disk}}(p^{[i]}; Q_\delta) \cap B(\frac{1}{2}(p^{[i]} + p^{[j]}), r)$ is shown in Figure 4.3(a).

The lightly and darkly shaded regions together represent X_{temp} at the current iteration. The darkly shaded region represents $X_{\text{temp}} \cap H_{Q_\delta}(v)$, where v is as described in step 3:. The outcome of the execution is shown in Figure 4.3(b).

Next, we characterize the main properties of the iterated truncation algorithm. It is convenient to define the set

$$J = \{(p, q) \in Q_\delta \times Q_\delta \mid [p, q] \in Q_\delta \text{ and } \|p - q\|_2 \leq r\}.$$

Proposition 4.9 (Properties of the iterated truncation algorithm). *Consider the δ -contraction of a compact allowable environment Q_δ with κ strict concavities, and let $(p^{[i]}, p^{[j]}) \in J$. The following statements hold:*

- (i) *The iterated truncation algorithm, invoked with arguments $(p^{[i]}, p^{[j]}; Q_\delta)$, terminates in at most κ steps; denote its output by $X_{\text{vis-disk}}(p^{[i]}, p^{[j]}; Q_\delta)$.*
- (ii) *$X_{\text{vis-disk}}(p^{[i]}, p^{[j]}; Q_\delta)$ is nonempty, compact and convex.*
- (iii) *$X_{\text{vis-disk}}(p^{[i]}, p^{[j]}; Q_\delta) = X_{\text{vis-disk}}(p^{[j]}, p^{[i]}; Q_\delta)$.*
- (iv) *The set-valued map $(p, q) \mapsto X_{\text{vis-disk}}(p, q; Q_\delta)$ is closed at J .*

In the interest of brevity, we do not include the proof here and instead refer the reader to [Ganguli et al. \(2009\)](#). We just mention that fact (iii) is a consequence of the fact that all relevant concavities in the computation of $X_{\text{vis-disk}}(p^{[i]}, p^{[j]}; Q_\delta)$ are visible from both agents $p^{[i]}$ and $p^{[j]}$. We are finally ready to provide analogs of Definition 4.4 and Lemma 4.5.

Defnition 4.10 (Line-of-sight connectivity constraint set). Consider a nonconvex allowable environment Q_δ and two agents i and j within range-limited line of sight. We call $\text{vis-dist}(p^{[i]}, p^{[j]}; Q_\delta)$ the *pairwise line-of-sight connectivity constraint set* of agent i with respect to agent j . Furthermore, given agents at positions $\mathbf{P} = \{p^{[1]}, \dots, p^{[n]}\}$ that are all within range-limited line of sight of agent i , the *line-of-sight connectivity constraint sets*

of agent i with respect
to $\mathbf{P}$ is

[i]

[i]

[i]

$$X_{\text{vis-disk}}(p, \mathbf{P}; Q_\delta) = \{x \in X_{\text{vis-disk}}(p, q; Q_\delta) \mid q \in \mathbf{P} \setminus \{p\}\}.$$

The following result is a consequence of Proposition 4.9.

Lemma 4.11 (Maintaining network line-of-sight connectivity). *Consider a group of agents at positions $(\ell) \mathbf{P} p^{[1]}(\ell), \dots, p^{[n]}(\ell) \in Q_\delta$ at time ℓ . If each agent's control $u^{[i]}(\ell)$ takes value in*

$$u^{[i]}(\ell) \in X_{\text{vis-disk}}(p^{[i]}(\ell), \mathbf{P}(\ell); Q_\delta - p^{[i]}(\ell)), \quad i \in \{1, \dots, n\},$$

then, according to the discrete-time motion model (4.1.1):

(i) each agent remains in its constraint set, that is,

$$p^{[i]}(\ell + 1) \in X_{\text{vis-disk}}(p^{[i]}(\ell), \mathbf{P}(\ell); Q_\delta);$$

(ii) each edge of $G_{\text{vis-disk}, Q_\delta}$ at $\mathbf{P}(\ell)$ is maintained after the motion step, that is, if $p^{[i]}$ and $p^{[j]}$ are within range-limited line of sight at time ℓ , then they are within range-limited line of sight also at time $\ell + 1$;

(iii) if $G_{\text{vis-disk}, Q_\delta}$ at $\mathbf{P}(\ell)$ is connected, then $G_{\text{vis-disk}, Q_\delta}$ at $\mathbf{P}(\ell + 1)$ is connected; and

(iv) the number of connected components of the graph $G_{\text{vis-disk}, Q_\delta}$ at $\mathbf{P}(\ell + 1)$ is equal to or smaller than the number of connected components of the graph $G_{\text{vis-disk}, Q_\delta}$ at $\mathbf{P}(\ell)$.

Remark 4.12 (Constraints for relative-sensing networks: cont'd). Following up on Remarks 4.3 and 4.6, we consider a relative-sensing network with range-limited visibility sensors (see Example 3.16). To compute the connectivity constraint set for this network, it suffices to provide a relative sensing version of the iterated truncation algorithm:

function relative-sensing iterated truncation(y ;
 y_{env})

% Executed by robot i with range-limited visibility sensor:

robot measurement is $y = p^{[j]} \in \text{Vi}_{\text{disk}}(0_2; (Q_\delta)_i)$ for $j \neq i$

environment measurement is $y_{\text{env}} = \text{Vi}_{\text{disk}}(0_2; (Q_\delta)_i)$

1: $\mathbf{X}_{\text{temp}} := y_{\text{env}} \cap \overline{B}_{\frac{\rho_L}{2}, \frac{r_L}{2}}$

2: while $\partial \mathbf{X}_{\text{temp}}$ contains a concavity do

3: $v :=$ a strictly concave point of $\partial \mathbf{X}_{\text{temp}}$ closest to $[0_2, y]$

4: $\mathbf{X}_{\text{temp}} := \mathbf{X}_{\text{temp}} \cap H_{y_{\text{env}}}(v)$

5: return $\mathbf{X}_{\text{temp}}$

The algorithm output is $\mathbf{X}_{\text{vis-disk}}(0_d, y)_i$ for $y = p^{[j]} \in \text{Vi}_{\text{disk}}(0_2; (Q_\delta)_i)$.

Next, we relax the constraints in Definition 4.10 to

provide the network 186

nodes with larger, and therefore less conservative, motion constraint sets. Similarly to Section 4.2.2, we seek to enforce the preservation of a smaller number of range-limited line-of-sight links, while still making sure that the overall network connectivity is preserved. To do this, we recall from Section 2.2 the notion of locally cliqueless graph $G_{lc,G}$ of a proximity graph G .

This proximity graph is illustrated in Figure 2.12. Let us use the short-hand notation $G_{lc-vis-disk,Q_\delta} \equiv G_{lc,G_{vis-disk,Q_\delta}}$. From Theorems 2.11(ii) and (iii), respectively, recall that $G_{lc-vis-disk,Q_\delta}$ has the following properties:

- (i) it has the same connected components as $G_{vis-disk,Q_\delta}$, that is, for all point sets $P \subset \mathbb{R}^d$, the graph has the same number of connected components consisting of the same vertices; and
- (ii) it is spatially distributed over $G_{vis-disk,Q_\delta}$.

Because of (i), to maintain or decrease the number of connected components of a range-limited visibility graph, it is sufficient to maintain or decrease the number of connected components of the sparser graph $G_{lc-vis-disk,Q_\delta}$. Because of (ii), it is possible for each agent to determine which of its neighbors in $G_{vis-disk,Q_\delta}$ are its neighbors also in $G_{lc-vis-disk,Q_\delta}$. We formalize this discussion as follows.

Definition 4.13 (Locally cliqueless line-of-sight connectivity constraint set). Consider a nonconvex allowable environment $Q_\delta \subset \mathbb{R}^2$ and a group of agents at positions $P = \{p^{[1]}, \dots, p^{[n]}\} \subset Q$. The *locally cliqueless line-of-sight connectivity constraint set* of agent i with respect to P is

$$X_{lc-vis-disk}(p^{[i]}, P; Q_\delta) = \{q \in X_{vis-disk}(p^{[i]}, q; Q_\delta) \mid$$

$$q \in P \text{ s.t. } (q, p^{[i]}) \text{ is an edge of } G_{lc-vis-disk,Q_\delta}(P)\}.$$

The following result is a direct consequence of the previous arguments.

Lemma 4.14 (Maintaining connectivity of sparser networks). *Consider a group of agents at positions $\{p^{[1]}(\ell), \dots, p^{[n]}(\ell)\} \subseteq \mathbb{R}^2$ at time ℓ . If each agent's control $u^{[i]}(\ell)$ takes value in*

$$u^{[i]}(\ell) \in X_{\text{lc-vis-disk}}(p^{[i]}(\ell), P(\ell); Q_\delta - p^{[i]}(\ell)), \quad i \in \{1, \dots, n\},$$

then, according to the discrete-time motion model (4.1.1):

- (i) *each agent remains in its locally cliqueless line-of-sight connectivity constraint set;*
- (ii) *two agents that are in the same connected component of $G_{\text{lc-vis-disk}, Q_\delta}$ remain at the same connected component after the motion step; and*
- (iii) *the number of connected components of the graph $G_{\text{lc-vis-disk}, Q_\delta}$ at*

$P(\ell+1)$ is equal to or smaller than the number of connected components of the graph $G_{lc-vis-disk, Q_\delta}$ at $P(\ell)$.

4.3 RENDEZVOUS ALGORITHMS

In this section, we present some algorithms that might be used by a robotic network to achieve rendezvous. Throughout the section, we mainly focus on the uniform network disk of locally connected first-order agents in $\mathbb{R}^d$; this robotic network was introduced in Example 3.4.

4.3.1 Averaging control and communication law

We first study a behavior in which agents move toward a position computed as the average of the received messages. This law is related to the distributed linear algorithms discussed in Section 1.6 and, in particular, to adjacency-based agreement algorithms and Vicsek's model. This algorithm has also been studied in the context of "opinion dynamics under bounded confidence" and is known in the literature as the Krause model.

We loosely describe the averaging law, which we denote by $\mathcal{C}_{\text{averaging}}$, as follows:

[Informal description] In each communication round each agent performs the following tasks: (i) it transmits its position and receives its neighbors' positions; (ii) it computes the average of the point set comprised of its neighbors and of itself. Between communication rounds, each robot moves toward the average point that it computed.

We next formulate the algorithm, using the description

model of Chapter 3. The law¹ is uniform, static, and data-sampled, with standard message-generation function:

Robotic Network: disk with motion model (4.1.1) in $\mathbb{R}^d$,
 with absolute sensing of own position, and
 with communication range r

Distributed Algorithm: averaging

¹From Definition 3.9 and Remark 3.11 recall that a control and coordination law (1) is uniform if processor state set, message-generation, state-transition and control functions are the same for each agent; (2) is static if the processor state set is a singleton, i.e., the law requires no memory;

(3) is data-sampled if the control functions are independent of the current position of the robot and depend only upon the robots position at the last sample time.

Alphabet: $\mathbf{A} = \mathbf{R}^d \cup \{\text{null}\}$

function $\text{msg}(p, i)$

1: return p

function $\text{ctl}(p, y)$

1: return $\text{avrg}(\{p\} \cup \{p_{\text{rcvd}} \mid p_{\text{rcvd}} \text{ is a non-null message in } y\}) - p$

An implementation of this control and communication law is shown in Figure 4.5 for $d = 1$. Note that, along the evolution, (1) several robots *rendezvous*, that is, agree upon a common location, and (2) some robots are connected at the simulation's beginning and not connected at the simulation's end (e.g., robots number 8 and 9, counting from the left). Our analysis

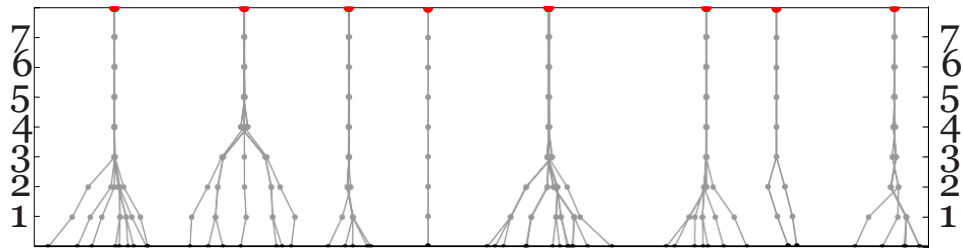


Figure 4.5 The evolution of a robotic network $\mathbf{S}_{\text{disk}}$, with $r = 1.5$, under the averaging control and communication law. The vertical axis corresponds to the elapsed time, and the horizontal axis to the positions of the agents in the real line. The 51 agents are initially randomly deployed over the interval $[-15, 15]$.

of the performance of this law is contained in the following theorem, whose proof is postponed to Section 4.6.1.

Theorem 4.15 (Correctness and time complexity of averaging law). *For $d = 1$, the network $\mathbf{S}_{\text{disk}}$, the law averaging achieves the task rndzvs with time complexity*

$$\text{TC}(\text{T}_{\text{rndzvs}}, \text{CC}_{\text{averaging}}) \in O(n^5),$$

$$\text{TC}(\text{T}_{\text{rndzvs}}, \text{CC}_{\text{averaging}}) \in \Omega(n).$$

4.3.2 Circumcenter control and communication laws

Here, we define the `crcmcntr` control and communication law for the net-work_{disk}. The law solves the rendezvous problem while keeping the net-work connected. This law was introduced by [Ando et al. \(1999\)](#) and later studied by [Lin et al. \(2007a\)](#) and [Cortés et al. \(2006\)](#).

We begin by recalling two useful geometric concepts: (i) given a bounded set S , its circumcenter $\text{CC}(S)$ is the center of the closed ball of minimum radius containing S (see Section 2.1.3); (ii) given a point p in a convex set Q and a second point q , the from-to-inside map $\text{fti}(p, q, S)$ is the point in the closed segment $[p, q]$ which is at the same time closest to q and inside S (see Section 2.1.1). Finally, recall also the connectivity constraint set introduced in Definition 4.4.

We loosely describe the `crcmctr` law ζ denoted by `crcmctr`, as follows:

[Informal description] In each communication round each agent performs the following tasks: (i) it transmits its position and receives its neighbors' positions; (ii) it computes the circumcenter of the point set comprised of its neighbors and of itself. Between communication rounds, each robot moves toward this circumcenter point while maintaining connectivity with its neighbors using appropriate connectivity constraint sets.

We next formulate the algorithm, using the description model of Chapter 3. The law is uniform, static, and data-sampled, with standard message-generation function:

Robotic Network: disk with discrete-time motion model (4.1.1), S with absolute sensing of own position, and with communication range r , in $\mathbb{R}^d$

Distributed Algorithm: `crcmctr`

Alphabet: $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$

```

function msg( $p, i$ )

1: return  $p$ 

function ctl( $p, y$ )

1:  $p_{\text{goal}} := \text{CC}(\{p\} \cup \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\})$ 
2:  $X := X_{\text{disk}}(p, \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\})$ 
3: return fti( $p, p_{\text{goal}}, X$ ) -  $p$ 

```

This algorithm is illustrated in Figure 4.6.

Next, let us note that it is possible and straightforward to implement the circumcenter law as a static relative-sensing control law on the relative-sensing network with disk_{disk} introduced in Example 3.15: sensors S^{rs}

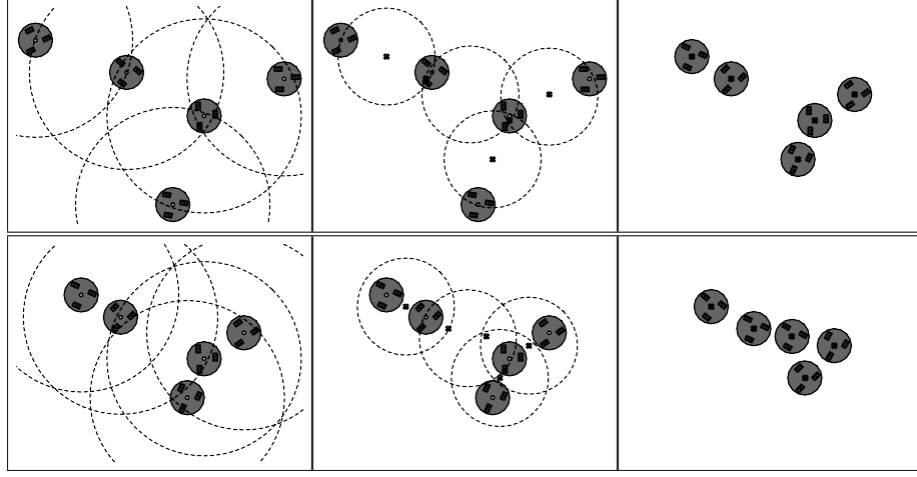


Figure 4.6 An illustration of the execution of the `crmcctr` algorithm. Each row of plots represents an iteration of the law. In each round, each agent computes its goal point and its constraint set, and then moves toward the goal while remaining in the constraint set.

Relative Sensing Network: S^{rs} with motion model (4.1.2),
no communication, relative sensing for robot i given by:
robot measurements y contains $p^{[j]} \in B(0_2, r)$ for all $j \neq i$

Distributed Algorithm: relative-sensing `crmcctr`

function `ctl(y)`

- 1: $p_{\text{goal}} := \text{CC}(\{0_d\} \cup \{p_{\text{snsd}} \mid \text{for all non-null } p_{\text{snsd}} \in y\})$
 - 2: $X := X_{\text{disk}}(0_d, \{p_{\text{snsd}} \mid \text{for all non-null } p_{\text{snsd}} \in y\})$
 - 3: return `fti`($0_d, p_{\text{goal}}, X$)
-

In the remainder of this section, we generalize the circumcenter law in a number of ways: (i) we modify the constraint set by imposing bounds on the control inputs and by relaxing the connectivity constraint as much as possible, while maintaining connectivity guarantees; and (ii) we implement the circumcenter law on two distinct communication graphs. Let us note that many of these generalized circumcenter laws can also be implemented as relative-sensing control laws; in the interest of brevity, we do not present the details.

191

832

4.3.2.1 Circumcenter law with control bounds and relaxed connectivity constraints

First, assume that the agents have a compact input space $U = B(0_d, u_{\max})$, with $u_{\max} \in \mathbb{R}_{>0}$. Additionally, we adopt the relaxed G -connectivity constraint sets as follows. Let G be a proximity graph that is spatially distributed over $G_{\text{disk}}(r)$ and that has the same connected components as $G_{\text{disk}}(r)$; examples include $G_{\text{RN}} \cap G_{\text{disk}}(r)$, $G_{\text{disk}}(r)$, and $G_{\text{LD}}(r)$. Recall the G -connectivity constraint set from Definition 4.7. Combining the relaxed connectivity constraint and the control magnitude bound, we redefine the control function in the `crcmctr` law to be:

```
function ctl(p, y)
% Includes control bound and relaxed G-connectivity constraint
1:  $p_{\text{goal}} := \text{CC}(\{p\} \cup \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\})$ 
   2:  $X := X_{\text{disk}, G}(p, \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}) \cap B(p, u_{\max})$ 
3: return fti(p,  $p_{\text{goal}}$ ,  $X$ ) - p
```

Second, the circumcenter law can be implemented also on robotic networks with different proximity graphs. For example, we can implement the circumcenter algorithm without any change on the following network.

4.3.2.2 Circumcenter law on the limited Delaunay graph

We consider the same set of physical agents as in S_{disk} . For $r \in \mathbb{R}_{>0}$, we adopt as communication graph the r -limited Delaunay graph $G_{\text{LD}}(r)$,

described in Section 2.2. These data define the uniform robotic network $\text{net}_{\text{LD}} = (1, \dots, n, \mathcal{G}, E_{\text{LD}})$, as described in Example 3.4. On this network, we implement the `crcmctr` law without any change, that is, with the same message-generation and control function as we did for the implementation on the network S_{disk} .

4.3.2.3 Parallel circumcenter law on the ∞ -disk graph

We consider the network $S_{\infty\text{-disk}}$ of first-order robots in $\mathbb{R}^d$, connected according to the $G_{\infty\text{-disk}}(r)$ graph (see Example 3.4). For this network, we define the `pll-crcmctr` law, which we denote by `pll-crcmctr`, by designing d decoupled circumcenter laws running in parallel on each coordinate axis of $\mathbb{R}^d$. As before, this law is uniform and static. What is remarkable, however, is that no constraint is required to maintain connectivity (see Exercise E4.4).

The parallel circumcenter of the set S , denoted by $\text{PCC}(S)$, is the center of the smallest axis-aligned rectangle containing S . In other words, $\text{PCC}(S)$ is the component-wise circumcenter of S (see Figure 4.7). We state the

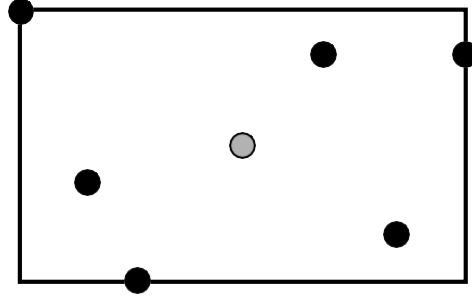


Figure 4.7 The gray point is the parallel circumcenter of the collection of black points.

parallel circumcenter law as follows:

Robotic Network: S with discrete-time ∞ -disk motion model (4.1.1) in $\mathbb{R}^d$, with absolute sensing of own position, and with communication range r in L^∞ -metric

Distributed Algorithm: pll-crcmctr

Alphabet: $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$

function $\text{msg}(p, i)$

1: return p

function $\text{ctl}(p, y)$

1: $p_{\text{goal}} := \text{PCC}(\{p\} \cup \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\})$

2: return $p_{\text{goal}} - p$

4.3.3 Correctness and complexity of circumcenter laws

In this section, we characterize the convergence and complexity properties of the circumcenter law and of its

variations. The following theorem summarizes the results known in the literature about the asymptotic properties of the circumcenter law.

Theorem 4.16 (Correctness of the circumcenter laws). *For $d \in \mathbf{N}$,*

$r \in \mathbf{R}_{>0}$, and $\varepsilon \in \mathbf{R}_{>0}$, the following statements hold:

- (i) on the network S_{disk} , the law $CC_{\text{circumcenter}}$ (with control magnitude bounds and relaxed G-connectivity constraints) achieves the exact rendezvous task T_{rdzvs} ;
- (ii) on the network S_{LD} , the law $CC_{\text{circumcenter}}$ achieves the ε -rendezvous task $T_{\varepsilon\text{-rdzvs}}$; and
- (iii) on the network $S_{\infty\text{-disk}}$, the law $CC_{\text{pll-circumcenter}}$ achieves the exact rendezvous task T_{rdzvs} .

Furthermore, the evolutions of $(S_{\text{disk}}, CC_{\text{circumcenter}})$, $(S_{\text{LD}}, CC_{\text{circumcenter}})$, and

$(S_{\infty\text{-disk}}, CC_{\text{pll-circumcenter}})$ have the following properties:

- (iv) If any two agents belong to the same connected component of the respective communication graph at $\ell \in \mathbb{Z}_{\geq 0}$, then they continue to belong to the same connected component for all subsequent times $k \geq \ell$.
- (v) For each evolution, there exists $P^* = (p_1^*, \dots, p_n^*) \in (\mathbb{R}^d)^n$ such that:
 - (a) the evolution asymptotically approaches P^* ; and
 - (b) for each $i, j \in \{1, \dots, n\}$, either $p_i^* = p_j^*$, or $\|p_i^* - p_j^*\|_2 > r$ (for the networks S_{disk} and S_{LD}) or $\|p_i^* - p_j^*\|_\infty > r$ (for the network $S_{\infty\text{-disk}}$).

The proof of this theorem is given in Section 4.6.2. The robustness of the circumcenter control and communication laws can be characterized with respect to link failures (see Cortés et al., 2006).

Next, we analyze the time complexity of $CC_{\text{circumcenter}}$. As we will see, next, the complexity of $CC_{\text{circumcenter}}$ differs dramatically when applied to robotic networks with different communication graphs. We provide complete results for the networks S_{disk} and S_{LD} only for the case $d = 1$.

Theorem 4.17 (Time complexity of circumcenter laws). *For $r \in \mathbb{R}_{>0}$ and $\varepsilon \in]0, 1[$, the following statements hold:*

- (i) on the network S_{disk} , evolving on the real line $\mathbb{R}$ (i.e., with $d = 1$), $TC(T_{\text{rdzvs}}, CC_{\text{circumcenter}}) \in \Theta(n)$;
- (ii) on the network S_{LD} , evolving on the real line $\mathbb{R}$ (i.e., with $d = 1$),

$\text{TC}(\mathcal{T}_{(r_\varepsilon)\text{-rndzvs}}, \mathcal{CC}_{\text{crcmctr}}) \in \Theta(n^2 \log(n\varepsilon^{-1}));$ and

(iii) on the network $S_{\infty\text{-disk}}$, evolving on Euclidean space (i.e., with $d \in \mathbb{N}$), $\text{TC}(\mathcal{T}_{\text{rndzvs}}, \mathcal{CC}_{\text{pll-crcmctr}}) \in \Theta(n)$.

The proof of this result is contained in [Martínez et al. \(2007b\)](#).

Remark 4.18 (Analysis in higher dimensions). The results in Theorems 4.17(i) and (ii) induce lower bounds on the time complexity of the circumcenter law in higher dimensions. Indeed, for arbitrary $d \geq 1$, we have the following:

- (i) on the network S_{disk} , $\text{TC}(T_{\text{rndzvs}}, CC_{\text{circmctr}}) \in \Omega(n)$;
- (ii) on the network S_{LD} , $\text{TC}(T_{(r/\varepsilon)\text{-rndzvs}}, CC_{\text{circmctr}}) \in \Omega(n^2 \log(n\varepsilon^{-1}))$.

We have performed extensive numerical simulations for the case $d = 2$ and the network S_{disk} . We run the algorithm starting from generic initial configurations (where, in particular, the robots' positions are not aligned) contained in a bounded region of $\mathbb{R}^2$. We have consistently obtained that the time complexity to achieve T_{rndzvs} with CC_{circmctr} starting from these initial configurations is independent of the number of robots. This leads us to conjecture that initial configurations where all robots are aligned (equivalently, the 1-dimensional case) give rise to the worst possible performance of the algorithm. In other words, we conjecture that, for $d \geq 2$, $\text{TC}(T_{\text{rndzvs}}, CC_{\text{circmctr}}) = \Theta(n)$.

Remark 4.19 (Congestion effects). As discussed in Remark 3.8, one way of incorporating congestion effects into the network operation is to assume that the parameters of the physical components of the network depend upon the number of robots—for instance, by assuming that the communication range decreases with the number of robots. Theorem 4.17 presents an alternative, equivalent, way of looking at congestion: the results hold under the assumption that the communication range is constant, but allow for the diameter of the initial network configuration (the maximum inter-agent distance) to grow unbounded with the number of robots.

4.3.4 The circumcenter law in nonconvex environments

In this section, we adapt the circumcenter algorithm to

work on networks in planar nonconvex allowable environments. Throughout the section, we only consider the case of a compact allowable nonconvex environment Q contracted into Q_δ for a small positive δ . We present the algorithm in two formats: for the communication-based network $S_{\text{vis-disk}}$ described in Example 3.6, and for the relative-sensing network S^{rs} described in Example 3.16.

We modify the circumcenter algorithm in three ways: first, we adopt the connectivity constraints described in the previous section for range-limited line-of-sight links; second, we further restrict the robot motion to remain inside the relative convex hull of the sensed robot positions; and third, we

move towards the circumcenter of the constraint set,
instead of the circum- center of the neighbors positions.
The details of the algorithm are as follows:

Robotic Network: $S_{\text{vis-disk}}$ with discrete-time motion
model (4.1.1), absolute sensing of own
position and of Q_δ , and communication
range r within line of sight ($G_{\text{vis-disk}, Q_\delta}$)

Distributed Algorithm: nonconvex crcmctr

Alphabet: $\mathbf{A} = \mathbf{R}^2 \cup \{\text{null}\}$

function msg(p, i)

1: return p

function ctl(p, y)

1: $X_1 := X_{\text{vis-disk}}(p, \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}; Q_\delta)$
2: $X_2 := \text{rco}(\{p\} \cup \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}; \text{Vi}(p; Q_\delta))$
3: $p_{\text{goal}} := \text{CC}(X_1 \cap X_2)$
4: return fti($p, p_{\text{goal}}, B(p, u_{\text{max}}) - p$)

Next, we present the relative sensing version; recall that $p^{[i]} = 0_2$ and that, as discussed in Section 3.2.3 in the context of the evolution of a relative sensing network with environment sensors, y_{env} denotes the environment measurement provided by the range-limited visibility sensor:

Relative Sensing Network: S^{rs} with motion model (4.1.2) in

Q_δ ,
no communication, relative sensing for robot i given by:
robot measurements y contains $p^{[j]} \in \text{Vi}_{\text{disk}}^{\text{vis-disk}}(0_2; (Q_\delta)_i)$ for j
 $f = i$

environment sensing is $y_{\text{env}} = \text{Vi}_{\text{disk}}(0_2; (Q_\delta)_i)$

Distributed Algorithm: nonconvex relative-

sensing crcmctr

function ctl(y, y_{env})

$1: X_1 := X_{\text{vis-disk}}(0_2, \{p_{\text{snsd}} \mid \text{for all non-null } p_{\text{snsd}} \in y\}; y_{\text{env}})$ $2: X_2 := \text{rco}(\{0_2\} \cup \{p_{\text{snsd}} \mid \text{for all non-null } p_{\text{snsd}} \in y\}; y_{\text{env}})$ $3: p_{\text{goal}} := \text{CC}(X_1 \cap X_2)$
 $4: \text{return fti}(0_2, p_{\text{goal}}, \overline{B(0_2, u_{\text{max}})})$

Theorem 4.20 (Correctness of the circumcenter law in nonconvex environments). *For $\delta > 0$, let Q_δ be a contraction of a compact allowable nonconvex environment Q . For $r \in \mathbb{R}_{>0}$ and $\varepsilon \in \mathbb{R}_{>0}$, on the network $S_{\text{vis-disk}}$, the law $\text{CC}_{\text{nonconvex circumctr}}$ (with control magnitude bounds) achieves the ε -rendezvous task $\varepsilon\text{-rndzvs}$. Furthermore, the evolution has the following properties:*

T

- (i) If any two agents belong to the same connected component of the graph $G_{\text{vis-disk}, Q_\delta}$ at $\ell \in \mathbb{Z}_{\geq 0}$, then they continue to belong to the same connected component for all subsequent times $k \geq \ell$.
- (ii) There exists $P^* = (p_1^*, \dots, p_n^*) \in Q^n$ such that:
 - (a) the evolution asymptotically approaches P^* ; and
 - (b) for each $i, j \in \{1, \dots, n\}$, either $p_i^* = p_j^*$, or p_i^* and p_j^* are not within range-limited line of sight.

The proof of this result can be found in Ganguli et al. (2009). A brief sketch of the proof steps is presented in Section 4.6.4. The complexity of the nonconvex `crcmctr` law has not been characterized. However, note that the evolution from any initial configuration such that G_{vis, Q_δ} is complete is also an evolution of the `crcmctr` law discussed in Section 4.3.2, and hence Theorem 4.17(i) induces a lower bound on the time complexity.

4.4 SIMULATION RESULTS

In this section, we illustrate the execution of some circumcenter control and communication laws introduced in this chapter.

The `crcmctr` law is im-

plemented on the networks S_{disk} , S_{LD} , and $S_{\infty\text{-disk}}$ in Mathematica[®] as a

library of routines and a main program running the simulation. The pack-

ages `PlanGeom.m` and `SpatialGeom.m` contain routines for the computation of geometric objects in $\mathbb{R}^2$ and $\mathbb{R}^3$, respectively. These routines are freely available at the book webpage <http://coordinationbook.info>

First, we show evolutions of $(\text{disk}, \text{crcmctr})$ in two and three dimensions in Figures 4.8 and 4.9, respectively. Measuring displacements in meters, we consider random initial positions over the square $[-7, 7] \times [-7, 7]$ and the cube $[-7, 7] \times [-7, 7] \times [-7, 7]$. The 25 robotic agents have a communication radius $r = 4$ and a compact input space $U = B(0_d, u_{\max})$, with $u_{\max} = 0.15$. As the simulations show, the task `rndzvs` is achieved,

as guaranteed by Theorem 4.16(i).

Second, within the same setup, we show an evolution of $(S_{LD}, \text{crcmcntr})$ in two dimensions in Figure 4.10. As the simulation shows, the task $\epsilon\text{-rndzvs}$ is achieved, as guaranteed by Theorem 4.16(ii).

Third, we show an evolution of $(S_{\infty\text{-disk}}, \text{pll-crcmcntr})$ in two dimensions in Figure 4.11. As the simulations show, the task rndzvs is achieved, as guaranteed by Theorem 4.16(iii).

Finally, we refer the interested reader to Ganguli et al. (2009) for simulation results for the nonconvex crcmcntr algorithm.

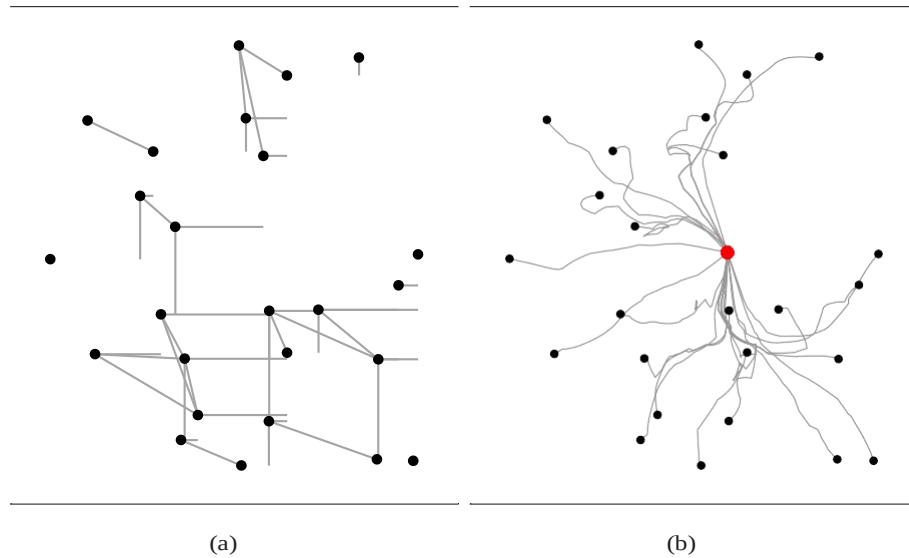


Figure 4.8 The evolution of $(S_{\text{disk}}, \text{circmctr})$ with $n = 25$ robots in 2 dimensions: (a) shows the initial connected network configuration; (b) shows the evolution of the individual agents until rendezvous is achieved.

4.5 NOTES

The rendezvous problem and the circumcenter algorithm were originally introduced by [Ando et al. \(1999\)](#). The circumcenter algorithm has been extended to other control policies, including asynchronous implementations, in [Lin et al. \(2007a,b\)](#). The circumcenter algorithm has been extended beyond planar problems to arbitrary dimensions in [Cortés et al. \(2006\)](#), where its robustness properties are also characterized. Regarding Theorem 4.16, the results on S_{disk} appeared originally in [Ando et al. \(1999\)](#); the results on S_{LD} and on $S_{\infty\text{-disk}}$ appeared originally in [Cortés et al. \(2006\)](#) and in [Martínez et al. \(2007b\)](#), respectively. Variations of the circumcenter law in the presence of noise and sensor errors are studied in [Martínez \(2009b\)](#). The continuous-time version of the circumcenter law, with no connectivity constraints, is analyzed in [Lin et al. \(2007c\)](#). Continuous-time control

laws for groups of robots with simple first-order dynamics and unicycle dynamics are proposed in [Lin et al. \(2004, 2005\)](#) and [Dimarogonas and Kyriakopoulos \(2007\)](#). In these works, the inter-robot topology is time dependent and assumed *a priori* to be connected at all times. Rendezvous under communication quantization is studied in [Fagnani et al. \(2004\)](#) and [Carli and Bullo \(2009\)](#). Rendezvous for unicycle robots with minimal sensing capabilities is studied by [Yu et al. \(2008\)](#). Relationships with classic curve-shortening flows are studied by [Smith et al. \(2007\)](#).

Rendezvous has also been studied within the computer science literature, 198

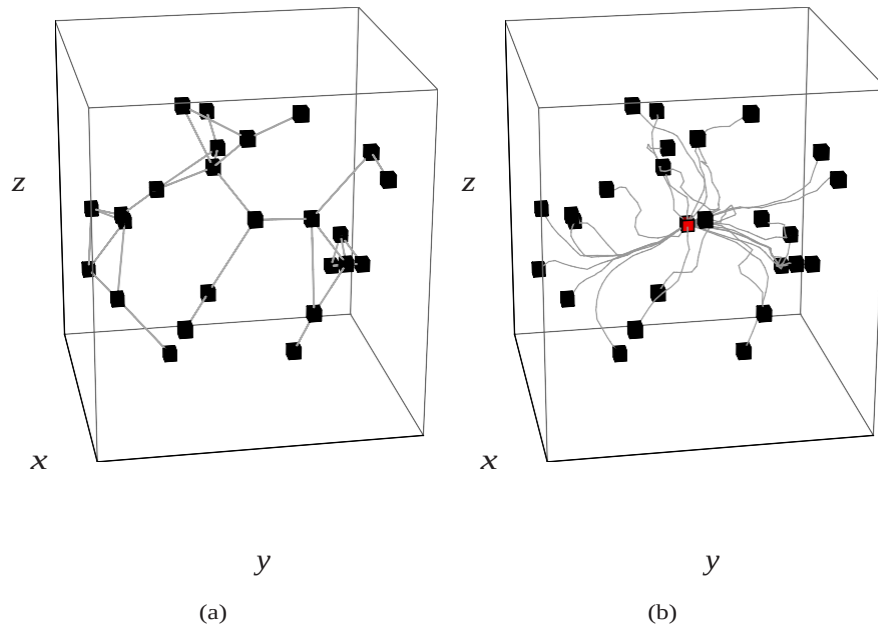


Figure 4.9 The evolution of $(S_{\text{disk}}, \text{crcmctr})$ with $n = 25$ robots in 3 dimensions: (a) shows the initial connected network configuration; (b) shows the evolution of the individual agents until rendezvous is achieved.

where the problem is referred to as the “gathering,” or point formation, problem. [Flocchini et al. \(1999\)](#) and [Suzuki and Yamashita \(1999\)](#) study the point formation problem under the assumption that each robot is capable of sensing all other robots. [Flocchini et al. \(2005\)](#) propose asynchronous algorithms to solve the gathering problem, and [Agmon and Peleg \(2006\)](#) study the solvability of the problem in the presence of faulty robots.

Multi-robot rendezvous with line-of-sight sensors is considered in [Roy and Dudek \(2001\)](#), where solutions are proposed based on the exploration of the unknown environment and the selection of appropriate rendezvous points at pre-specified times. [Hayes et al.](#)

(2003) also consider rendezvous at a specified location for visually guided agents, but the proposed solution requires each agent to have knowledge of the location of all other agents. The problem of computing a multi-robot rendezvous point in polyhedral surfaces made of triangular faces is considered in Lanthier et al. (2005). The perimeter-minimizing algorithm presented by Ganguli et al. (2009) solves the rendezvous problem for sensor-based networks with line-of-sight range-limited sensors in nonconvex environments.

Regarding the connectivity maintenance problem, a number of works have addressed the problem of designing a coordination algorithm that achieves a general, non-specified task while preserving connectivity. The centralized solution proposed in Zavlanos and Pappas (2005) allows for a general range of agent motions. The distributed solution presented by Savla et al. (2009b)

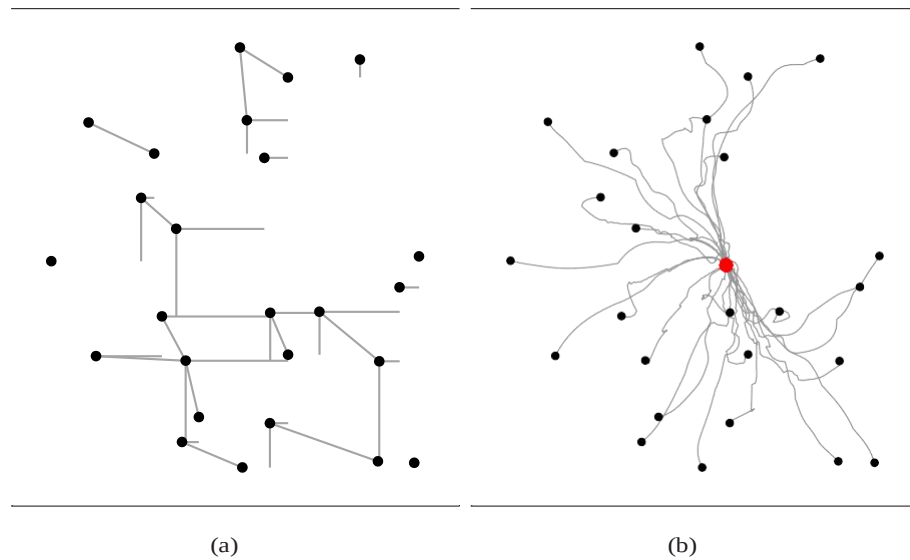


Figure 4.10 The evolution of $(S_{LD}, \text{crcmctr})$ with $n = 25$ robots in 2 dimensions: (a) shows the initial connected network configuration; (b) shows the evolution of the individual agents until rendezvous is achieved.

gives connectivity maintaining constraints for second-order control systems with input magnitude bounds. A distributed algorithm to perform graph rearrangements that preserve the connectivity is presented in [Schuresko and Cortés \(2007\)](#). Connectivity problems have been studied also in other contexts. [Langbort and Gupta \(2009\)](#) study the impact of the connectivity of the interconnection topology in a class of network optimization problems. [Spanos and Murray \(2005\)](#) generate connectivity-preserving motions between pairs of formations. [Ji and Egerstedt \(2007\)](#) design Laplacian-based control laws to solve formation control problems while preserving connectivity. Various works have focused on designing the network motion so that some desired measure of connectivity (e.g., algebraic connectivity) is maximized under position constraints. [Boyd \(2006\)](#) and [de Gennaro and Jad-babaie \(2006\)](#) consider convex constraints, while [Kim and Mesbahi](#)

(2006) deal with a class of nonconvex constraints. [Zavlanos and Pappas \(2007b\)](#) use potential fields to maximize algebraic connectivity.

A continuous-time version of the averaging control and communication law is also known as the Hegselmann-Krause model for “opinion dynamics under bounded confidence” (see [Hegselmann and Krause, 2002](#); [Lorenz, 2007](#)). In this model, each agent may change its opinion by averaging it with that of neighbors who are in an ε -confidence area. In other words, the difference between the agent’s opinion and those of its neighbors’ should be bounded by ε . A similar model where the communication between agents is random is the Deffuant-Weisbuch model, inspired by a model of dissemination of

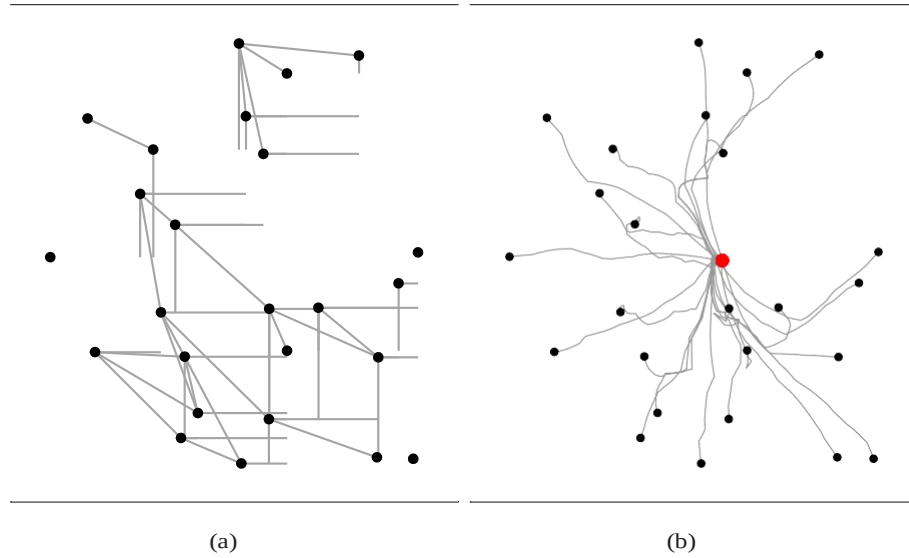


Figure 4.11 The evolution of $(S_{\infty\text{-disk}}, \text{pll-crcmcntr})$ with $n = 25$ robots in 2 dimensions: (a) shows the initial connected network configuration; (b) shows the evolution of the individual agents until rendezvous is achieved.

culture (see [Deffuant et al., 2000](#); [Axelrod, 1997](#)).

4.6 PROOFS

This section gathers the proofs of the main results presented in the chapter.

4.6.1 Proof of Theorem [4.15](#)

Proof. One can easily prove that, along the evolution of the network, the ordering of the agents is preserved, that is, the inequality $p^{[i]} \leq p^{[j]}$ is preserved at the next time step. However, links between agents are not necessarily preserved (see, e.g., Figure [4.8](#)). Indeed, connected components may split along the evolution. However, merging events do not occur. Consider two contiguous connected components C_1 and C_2 of $G_{\text{disk}}(r)$, with C_1 to the left of C_2 . By definition, the rightmost agent in the

component C_1 and the leftmost agent in the component C_2 are at a distance strictly larger than r . Now, by executing the algorithm, they can only but increase that distance, since the rightmost agent in C_1 will move to the left, and the leftmost agent in C_2 will move to the right. Therefore, connected components do not merge.

Consider first the case of an initial network configuration for which the communication graph remains connected throughout the evolution. Without

loss of generality, assume that the agents are ordered from left to right according to their identifier, that is, $p^{[1]}(0) \leq p^{[n]}(0)$. Let $\alpha_1, \dots, \alpha_n$ have the property that agents $2, \dots, \alpha_1$ are neighbors of agent 1, and agent α is not. (If, instead, all agents are within an interval of length r , then rendezvous is achieved in 1 time instant, and the statement in theorem is easily seen to be true.) Note that we can assume that agents $2, \dots, \alpha_1$ are also neighbors of agent α . If this is not the case, then those agents that are neighbors of agent 1 and not of agent α rendezvous with agent 1 at the next time instant. At the time instant $\ell = 1$, the new updated positions satisfy

$$p^{[1]}(1) = \frac{1}{\alpha - 1} \sum_{k=1}^{\alpha-1} p^{[k]}(0),$$

$$p^{[\gamma]}(1) \in \sum_{\alpha} \sum_{k=1}^{\gamma} p^{[k]}(0), *, \gamma \in \{2, \dots, \alpha - 1\},$$

where $*$ denotes a certain unimportant point.

Now, we show that

$$p^{[1]}(\alpha - 1) - p^{[1]}(0) = \frac{r}{\alpha(\alpha - 1)}. \quad (4.6.1)$$

Let us first show the inequality for $\alpha = 3$. Because of the assumption that the communication graph remains connected, agent 2 is still a neighbor of agent 1 at the time instant $\ell = 1$. Therefore, $p^{[1]}(2)$

$p^{[2]}(1))$, and from here we deduce

$$p^{[1]}(2) - p_{[1]}(0) \geq \frac{2_{[2]} - 1}{p} \quad (3)$$

[1]

$$(0)^{\sum} \geq \frac{1}{3} \cdot \frac{(0)^{+p} (0)^{+p}}{p^{[1]} p^{[2]}} (0)^{\sum} - p^{[1]} (0)^{\sum} \geq \frac{1}{6} p^{[3]} (0)^{\sum} - p^{[1]} (0)^{\sum} \geq \frac{r}{6}.$$

Let us now proceed by induction. Assume that inequality (4.6.1) is valid for $\alpha - 1$, and let us prove it for α . Consider first the possibility, when at the time instant $\ell = 1$, that the agent $\alpha - 1$ is still a neighbor of agent 1. In

this case, $p^{(2)} \geq \frac{1}{\alpha - 1} \sum_{k=1}^{\alpha} p^{(1)}$, and from here we deduce

$$\begin{aligned} \frac{p^{[1]}(2) - p^{[1]}(0)}{p^{[1]}(0)} &\geq \frac{1}{\alpha - 1} \cdot \frac{1}{\alpha} \sum_{k=1}^{\alpha} p^{[k]}(0) p^{[1]}(0) \\ &\geq \frac{1}{\alpha - 1} \cdot \frac{1}{\alpha} \sum_{k=1}^{\alpha} p^{[k]}(0) p^{[1]}(0) \\ &\geq \frac{1}{\alpha(\alpha - 1)} \cdot \frac{p^{[\alpha]}(0) - p^{[1]}(0)}{p^{[\alpha]}(0) - p^{[1]}(0)} \sum_{r=1}^{\alpha} p^{[r]}(0), \end{aligned}$$

which, in particular, implies (4.6.1). Consider then the case when agent $\alpha - 1$ is not a neighbor of agent 1 at the time instant $\ell = 1$. Let $\beta < \alpha$

such that agent $\beta - 1$ is a neighbor of agent 1 at $\ell = 1$, but agent β is not.

Since $\beta < \alpha$, we have by induction $p^{[1]}(\beta) = p^{[1]}(1) \cdot r$.
From here, we

$$\text{deduce that } p^{[1]}(\alpha - 1) = \frac{p^{[1]}(0) \cdot r^{\beta(\beta-1)}}{r^{\alpha(\alpha-1)}}.$$

It is clear that after $\ell_1 = \alpha - 1$, we could again consider two complementary cases (either agent 1 has all others as neighbors or not) and repeat the same argument once again. In that way, we would find ℓ_2 such that the distance traveled by agent 1 after ℓ_2 rounds would be lower bounded by $\frac{2^r}{n(n-1)}$.

Repeating this argument iteratively, the worst possible case is one in which

agent 1 keeps moving to the right and, at each time step, there is always another agent which is not a neighbor. Since the diameter of the initial condition P_0 is upper bounded by $(n - 1)r$, in the worst possible situation, ~~there exists~~ some time ℓ_k such that $\frac{(n-1)n}{k} = O(r(n - 1))$. This ~~implies that~~ $k = O((n - 1)^2 n)$. Now we can upper bound the total convergence time ℓ_k by $\ell_k = \sum_{i=1}^k \alpha_i \leq k(n - 1)$, where we have used that $\alpha_i \leq n$ for all $i = 1, \dots, n$. From here, we see that $\ell_k = O((n - 1)^3 n)$, and hence we deduce that in $O(n(n - 1)^3)$ time instants there cannot be any agent which

is not a neighbor of the agent 1. Hence, all agents rendezvous at the next time instant. Consequently,

$$\text{TC}(\text{T}_{\text{rndvzs}}, \text{CC}_{\text{averaging}}, P_0) = O(n(n - 1)^3).$$

Finally, for a general initial configuration P_0 , because there are a finite number of agents, only a finite number of splittings (at most $n - 1$) of the connected components of the communication graph can take place along the evolution. Therefore, we conclude that $\text{TC}(\text{T}_{\text{rndvzs}}, \text{CC}_{\text{averaging}}) = O(n^5)$.

Let us now prove the lower bound. Consider an initial configuration $P_0 \in \mathbb{R}^n$ where all agents are positioned in increasing order according to their identity, and exactly at a distance r apart—say, $p^{[i+1]}(0) - p^{[i]}(0) = r$,

$i = 1, \dots, n$
 Assume for simplicity that n is odd—when n is even, one can reason in an analogous way. Because of the symmetry of the initial condition, in the first time step, only agents 1 and n move. All the remaining agents remain in their position, because it coincides with the average of its neighbors' position and its own. At the second time step, only agents 1, 2, $n-1$, and n move, and the others remain static because of the symmetry. Applying this idea iteratively, one deduces that the time step when agents $\frac{n-1}{2}$ and $\frac{n+3}{2}$ move for the first time is lower bounded by $\frac{n-1}{2}$. Since both

agents have still at least a neighbor (agent $\frac{n+1}{2}$), the task T_{rndzvs} has not been achieved yet at this time step. Therefore, $\text{IC}(T_{\text{rndzvs}}, \text{CC}_{\text{averaging}}, P_0) \geq \frac{n-1}{2}$ and the result follows.

2

4.6.2 Proof of Theorem 4.16

Proof. We divide the proof of the theorem into three groups, one per network.

STEP 1: Facts on $(\mathbb{C}_{\text{disk}}, \text{CC}_{\text{crcmctr}})$. Fact S(iv) for $(\mathbb{C}_{\text{disk}}, \text{CC}_{\text{crcmctr}})$ is a direct consequence of the control function definition of the crcmctr law and Lemma 4.8.

Let us show fact (i). Because G has the same connected components as $G_{\text{disk}}(r)$, fact (iv) implies that the number of connected components of $G_{\text{disk}}(r)$ can only but decrease. In other words, the number of agents in each of the connected components of $G_{\text{disk}}(r)$ is non-decreasing. Since there is a finite number of agents, there must exist ℓ_0 such that the identity of the agents in each connected component of $G_{\text{disk}}(r)$ is fixed for all $\ell \geq \ell_0$ (that is, no more agents are added to the connected component afterwards). In what follows, without loss of generality, we assume that there is only one connected component after ℓ_0 , i.e., the graph is connected (if this is not the case, then the same argument follows through for each connected component).

We prove that the law $\text{CC}_{\text{crcmctr}}$ (with control magnitude bounds and relaxed-connectivity constraints) achieves the exact rendezvous task rndzvs in the following two steps:

- (a) We first define a set-valued dynamical system $((\mathbb{R}^d)^n, (\mathbb{R}^d)^n, T)$ such that the evolutions of $(S_{\text{disk}}, \text{CC}_{\text{crcmctr}})$, starting from an initial configuration where $G_{\text{disk}}(r)$ is connected, are contained in the set of evolutions of the set-valued dynamical system.
- (b) We then establish that any evolution of $((\mathbb{R}^d)^n, (\mathbb{R}^d)^n, T)$ converges to a point in $\text{diag}((\mathbb{R}^d)^n)$ (the point might be different for different evolutions).

This strategy is analogous to the discussion regarding the

Overapproximation Lemma for time-dependent systems in Section 1.3.5.

Let us perform (a). Given a connected graph tt with vertices $1, \dots, n$, let us consider the constraint sets and goal points defined with respect to tt . In other words, given $P = (p_1, \dots, p_n) \in (\mathbb{R}^d)^n$, define for each $i \in \{1, \dots, n\}$,

$$(p_{\text{goal}})_i := \text{CC}(\{p_i\} \cup \{p_j \mid j \in \mathbf{N}_{tt}(i)\}),$$

$$X_i := \bigcap_{\substack{p + p_j \\ r(P)}} B\left(\frac{p_i + p_j}{2}, \frac{r(P)}{2}\right) \mid j \in \mathbf{N}_{tt}(i) \cap B(p_i, u_{\max}),$$

where $r(P) = \max\{r, \max\{\|p_i - p_j\|_2 \mid j \in \mathbf{N}_{tt}(i)\}\}$. Since two neighbors

according to tt can be arbitrarily far from each other in $\mathbb{R}^d$, we need to modify the definition of the constraint set with the radius $r_i(P)$ to prevent X_i from becoming empty. Note that if $\|p_i - p_j\|_2 \leq r$ for all $j \in N_{tt}(i)$, then $r_i(P) = r$ and, therefore, $X_i = X_{\text{disk}, tt}(p_i, P) \cap B(p_i, u_{\max})$. It is also worth observing that both $(p_{\text{goal}})_i$ and X_i change continuously with $(p_1, \dots, p_n)$.

Define the map $\text{fti}_{tt} : (\mathbb{R}^d)^n \rightarrow (\mathbb{R}^d)^n$ by

$$\text{fti}_{tt}(p_1, \dots, p_n) = (\text{fti}(p_1, (p_{\text{goal}})_1, X_1), \dots, \text{fti}(p_n, (p_{\text{goal}})_n, X_n)).$$

One can think of fti_{tt} as a circumcenter law where the neighboring relationships among the agents never change. Because fti is continuous, and $(p_{\text{goal}})_i$ and X_i , $i \in \{1, \dots, n\}$, change continuously with $(p_1, \dots, p_n)$, we deduce that fti_{tt} is continuous.

We now define a set-valued dynamical system $((\mathbb{R}^d)^n, (\mathbb{R}^d)^n, T)$ through the set-valued map $T : (\mathbb{R}^d)^n \rightrightarrows (\mathbb{R}^d)^n$ given by

$$T(p_1, \dots, p_n) = \{\text{fti}_{tt}(p_1, \dots, p_n) \mid tt \text{ is a strongly connected digraph}\}.$$

Note that the evolution of the circumcenter law using a proximity graph such as $\text{disk}(r)$ is just one of the multiple evolutions described by this set-valued map. This concludes (a).

Let us now perform (b). To characterize the convergence properties of the set-valued dynamical system, we use the LaSalle Invariance Principle in Theorem 1.21. With the notation of this result, we select $W = (\mathbb{R}^d)^n$. This set is clearly strongly positively invariant for $((\mathbb{R}^d)^n, (\mathbb{R}^d)^n, T)$.

Closedness of the set-valued map. Since fti_{tt} is continuous for each digraph tt and there is a finite number of strongly connected digraphs on the vertices $\{1, \dots, n\}$, Exercise E1.9 implies that T is closed.

Common Lyapunov function. Define the function $V_{\text{diam}} : (\mathbb{R}^d)^n \rightarrow \mathbb{R}_{\geq 0}$ by

$$V_{\text{diam}}(P) = \max\{\|p_i - p_j\| \mid i, j \in \{1, \dots, n\}\}.$$

With a slight abuse of notation, we denote by $\text{co}(P)$ the convex hull of $p_1, \dots, p_n \in \mathbb{R}^d$. Note that $V_{\text{diam}}(P) = \text{diam}(\text{co}(P))$. The function V_{diam} has the following properties:

- (i) V_{diam} is continuous and invariant under permutations of its arguments.
- (ii) $V_{\text{diam}}(P) = 0$ if and only if $P \subseteq \text{diag}((\mathbb{R}^d)^n)$, where we recall that $\text{diag}((\mathbb{R}^d)^n) = \{(p_1, \dots, p_n) \in (\mathbb{R}^d)^n \mid p^{[i]} = p^{[n]} \in \mathbb{R}^d\}$ denotes the diagonal set of $(\mathbb{R}^d)^n$. This fact is an immediate consequence of the fact that, given a set $S \subset (\mathbb{R}^d)^n$, $\text{diam}(\text{co}(S)) = 0$ if

and only if S is a singleton.

(iii) V_{diam} is non-increasing along T on $(\mathbb{R}^d)^n$. Consider a finite set of points $S \in F((\mathbb{R}^d)^n)$ and let $\text{CC}(S)$ be its circumcenter. From Lemma 2.2(i), we have $\text{CC}(S) \in \text{co}(S)$. Therefore, for any strongly connected digraph tt , we have that $\text{co}(\text{fti}_{tt}(P)) \subset \text{co}(P)$ for any $P \in (\mathbb{R}^d)^n$. Since for any two sets $S_1, S_2 \subset (\mathbb{R}^d)^n$ such that $\text{co}(S_2) \subset \text{co}(S_1)$ it holds that $V_{\text{diam}}(S_2) \leq V_{\text{diam}}(S_1)$, then $V_{\text{diam}}(\text{fti}_{tt}(P)) \leq V_{\text{diam}}(P)$ for any strongly connected digraph tt , which implies that

V_{diam} is non-increasing along T on $(\mathbb{R}^d)^n$.

Bounded evolutions. Consider any initial condition $(p_1(0), \dots, p_n(0)) \in (\mathbb{R}^d)^n$. For any strongly connected digraph, tt , we have

$$\text{fti}_{tt}(p_1(\ell), \dots, p_n(\ell)) \in \text{co}(p_1(0), \dots, p_n(0)),$$

for all $\ell \in \mathbb{Z}_{\geq 0}$. Therefore, any evolution of the set-valued dynamical system $((\mathbb{R}^d)^n, (\mathbb{R}^d)^n, T)$ is bounded.

Characterization of the invariant set. By the LaSalle Invariance for set-valued dynamical systems in Theorem 1.21, any evolution with initial condition in $W = (\mathbb{R}^d)^n$ approaches the largest weakly positively invariant set M contained in

$$\{P \in (\mathbb{R}^d)^n \mid \exists P' \in T(P) \text{ such that } V_{\text{diam}}(P') = V_{\text{diam}}(P)\}.$$

We show that $M = \text{diag}((\mathbb{R}^d)^n)$. Clearly, $\text{diag}((\mathbb{R}^d)^n) \subset M$. To

prove the other inclusion, we reason by contradiction. Assume that $P \in M \setminus \text{diag}((\mathbb{R}^d)^n)$

and, therefore, $V_{\text{diam}}(P) > 0$. Let tt be a strongly connected di-

graph and consider $\text{fti}_{tt}(P)$. For each $i \in \{1, \dots, n\}$, we distinguish two cases

depending on whether p_i is or is not a vertex of $\text{co}(P)$. If $p_i \notin \text{Ve}(\text{co}(P))$, then Lemma 2.2(i) implies that $\text{fti}(p_i, (p_{\text{goal}})_i, X_i) \in \text{co}(P) \setminus \text{Ve}(\text{co}(P))$.

If $p_i \in \text{Ve}(\text{co}(P))$, then we must take into consideration the possibility of having more than one agent located at the same point. If the location of all the neighbors of i in the digraph tt coincides with p_i , then agent i will not move, and hence $\text{fti}(p_i, (p_{\text{goal}})_i, X_i) \in \text{Ve}(\text{co}(P))$. However, we can show that the application of

f_{tt} strictly decreases the number of agents located at p_i . Let us denote this number by N_i , that is,

$$N_i = |\{j \in \{1, \dots, n\} \mid p_j = p_i \text{ and } p_j \in \{p_1, \dots, p_n\}\}|.$$

Since the digraph tt is strongly connected, there must exist at least an agent located at p_i with a neighbor

which is not located at p_i (otherwise, all agents would be at p_i , which is a contradiction). In other words, there

exist $i_*, j \in \{1, \dots, n\}$ such that $p_{i_*} = p_i$, $p_j \neq p_i$, and $j \in \mathbf{N}_{tt}(i_*)$. By Lemma 2.2(i), we have that $(p_{\text{goal}})_{i_*} \in \text{co}(P) \setminus \text{Ve}(\text{co}(P))$ and, therefore,

$(p_{\text{goal}})_{i_*} \neq p_{i_*}$. Combining this with the fact that

$$\{p_i\} \cup \{p_j \mid j \in \mathbf{N}_{tt}(i)\} \subset \overline{B(p_{i_*}, r_{i_*}(P))},$$

we can apply Lemma 2.2(ii) to ensure that $]p_{i_*}, (p_{\text{goal}})_{i_*}[$ has nonempty intersection with X_{i_*} . Therefore, $\text{fti}(p_{i_*}, (p_{\text{goal}})_{i_*}, X_{i_*}) \in \text{co}(P) \setminus \text{Ve}(\text{co}(P))$, and the number N_i of agents located at p_i decreases at least by one with the application of fti_{tt} .

Next, we show that, after a finite number of steps, no agents remain at

the location p_i . Define $N = \max\{N_i \mid p_i \in \text{Ve}(\text{co}(P))\} < n - 1$. Then all agents in the configuration $\text{fti}_{tt_1}(\text{fti}_{tt_2}(\dots \text{fti}_{tt_N}(P)))$ are contained in $\text{co}(P) \setminus \text{Ve}(\text{co}(P))$, for any collection of strongly connected directed graphs $tt_1, \dots, tt_N$. Therefore, $\text{diam}(\text{co}(\text{fti}_{tt_1}(\text{fti}_{tt_2}(\dots \text{fti}_{tt_N}(P)))) < \text{diam}(\text{co}(P))$, which contradicts the fact that M is weakly invariant.

Point convergence. We have proved that any evolution of $((\mathbb{R}^d)^n, (\mathbb{R}^d)^n, T)$ approaches the set $\text{diag}((\mathbb{R}^d)^n)$. To conclude the proof, let us show that the convergence of each trajectory is to a point, rather than to the diagonal set.

Let $\{P(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$ be an evolution of the set-valued dynamical system. Since the sequence is contained in the compact set $\text{co}(P(0))$, there exists a convergent subsequence $\{P(\ell_k) \mid k \in \mathbb{Z}_{\geq 0}\}$, that is, there exists $p \in \mathbb{R}^d$ such

$$\lim_{k \rightarrow +\infty} P(\ell_k) = (p, \dots, p). \quad (4.6.2)$$

Let us show that the whole sequence $\{P(\ell) \mid \ell \in \mathbb{Z}_{\geq 0}\}$ converges to $(p, \dots, p)$.

Because of (4.6.2), for any $\varepsilon > 0$, there exists k_0 such that for $k \geq k_0$ one has

$$\text{co}(P(\ell_k)) \subset B(p, \varepsilon/n).$$

From this, we deduce that $\text{co}(P(\ell)) \subset B(p, \varepsilon/n)$

for all $\ell \geq \ell_{k_0}$, which in turn implies that $\|P(\ell) - (p, \dots, p)\|_2 \leq \varepsilon$ for all $\ell \geq \ell_{k_0}$, as claimed. This concludes (b).

The steps (a) and (b) imply that any evolution of $(S_{\text{disk}}, \mathcal{G})$

CC_{crcmctr}) starting from an initial configuration where $\text{disk}(r)$ is connected converges to a point in $\text{diag}((\mathbb{R}^d)^n)$. To conclude the proof of fact (i), we only need to establish that this convergence is in finite time. This last fact is a consequence of Exercise E4.5.

Fact (v) for $(S_{\text{disk}}, CC_{\text{crcmctr}})$ is a consequence of facts (i) and (iv).

STEP 2: Facts on $(S_{\text{LD}}, CC_{\text{crcmctr}})$. The proof of facts (i), (iv), and (v) for $(S_{\text{LD}}, CC_{\text{crcmctr}})$ is analogous to the proof of these facts for the pair $(S_{\text{disk}}, CC_{\text{crcmctr}})$, and we leave it to the reader.

STEP 3: Facts on $(S_{\infty\text{-disk}}, CC_{\text{pll-crcmctr}})$. From the expression for the control function of $CC_{\text{pll-crcmctr}}$, we deduce that the evolution un-

der $CC_{\text{pll-crcmctr}}$ of the robotic network $S_{\infty\text{-disk}}$ (in d dimensions) can be alternatively described as the evolution under CC_{crcmctr} of d robotic networks S_{disk} in $\mathbb{R}$ (see Exercise E4.4). Therefore, facts (i), (iv), and (v) for the pair $(S_{\infty\text{-disk}}, CC_{\text{pll-crcmctr}})$ follow from facts (i), (iv), and (v) for the pair $(S_{\text{disk}}, CC_{\text{crcmctr}})$.

□

4.6.3 Proof of Theorem 4.17

Proof. Let $P_0 = (p^{[1]}(0), \dots, p^{[n]}(0)) \in \mathbb{R}^n$ denote the initial condition.

Fact (i). For $d = 1$, the connectivity constraints on each agent $i \in \{1, \dots, n\}$ imposed by the constraint set

$$X_{\text{disk}}(p, \{p_{\text{rcvd}}^{[i]} \mid \text{for all non-null } p_{\text{rcvd}} \in {}^{[i]}y\}) \quad (4.6.3)$$

are superfluous. In other words, the goal configuration resulting from the evaluation by agent i of the control function of the crcmctr law belongs to the constraint set in (4.6.3). Moreover, the order of the robots on the real line is preserved from one time step to the next. Both observations are a consequence of Exercise E4.3.

Let us first establish the upper bound in fact (i). Consider the case when $\text{disk}(r)$ is connected at P_0 . Without loss of generality, assume that the agents are ordered from left to right according to their identifier, that is, $p^{[1]}(0) < \dots < p^{[n]}(0)$. Let $\alpha \in \{2, \dots, n\}$ have the property that agents $\alpha-1$ and α are neighbors of agent 1, and agent α is not. (If, instead, all agents are within an interval of length r , then rendezvous is achieved after one time step, and the upper bound in fact (i) is easily seen to be true.) Figure 4.12 presents an illustration of the definition of α . Note that we can

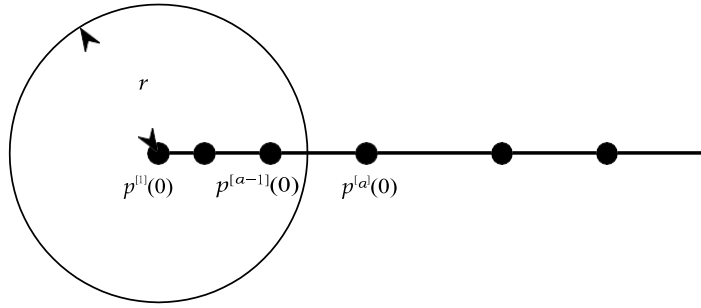


Figure 4.12 The definition of $\alpha \in \{3, \dots, n\}$ for an initial network configuration.

assume that agents $2, \dots, \alpha - 1$ are also neighbors of agent α . If this is not the case, then those agents that are neighbors of agent 1 and not of agent

α , rendezvous with agent 1 after one time step. At the time instant $\ell = 1$, the new updated positions satisfy

$$\begin{aligned} p^{[1]}(1) &= \frac{p^{[1]}(0) + p^{[\alpha]}(0)}{2}, \\ p^{[\gamma]}(1) &\in \frac{p^{[1]}(0) + p^{[\alpha]}(0)}{2}, \frac{p^{[1]}(0) + p^{[\gamma]}(0) + r}{2}, \end{aligned}$$

for $\gamma \in \{2, \dots, \alpha - 1\}$. These equalities imply that $p^{[1]}(1) - p^{[1]}(0) = \frac{1}{2}(p^{[\alpha]}(0) - p^{[1]}(0)) \leq \frac{1}{2}r$. Analogously, we deduce $p^{[1]}(2) - p^{[1]}(1) \leq \frac{1}{2}r$ and, therefore,

On the other hand, from (4.6.4), $p^{[1]}(2) - p^{[1]}(0) \leq r$. (where the symbol $*$ represents a certain unimportant point in

$$\begin{aligned} p^{[1]}(2) - p^{[1]}(0) &\geq \frac{1}{2}(p^{[1]}(1) + p^{[\alpha-1]}(1) - p^{[1]}(0)) \\ &\geq \frac{1}{2}(p^{[\alpha-1]}(1) - p^{[1]}(0)) \geq \frac{1}{2}(p^{[\alpha]}(0) - p^{[1]}(0)) \\ &= \frac{1}{4}(p^{[\alpha]}(0) - p^{[1]}(0)) \geq \frac{1}{4}r. \end{aligned} \quad (4.6.5)$$

Inequalities (4.6.4) and (4.6.5) mean that, after at most two time steps, agent 1 has traveled a distance greater than $r/4$. In turn, this implies that

$$\frac{1}{r} \text{diam}(\text{co}(P_0)) \leq \text{TC}(\text{T}_{\text{rendzvs}}, \text{CC}_{\text{cmentr}}, P_0) \leq \frac{4}{r} \text{diam}(\text{co}(P_0)).$$

If $\mathbb{G}_{\text{disk}}(r)$ is not connected at P_0 , note that along the network evolution, the connected components of the r -disk graph do not change. Using the previous characterization on the distance traveled by the leftmost

agent of each connected component in at most two time steps, we deduce that

$$\text{TC}(\mathbb{T}_{\text{ndzvs}}, \mathbb{CC}_{\text{rcmctr}}, P_0) \leq \frac{4}{r} \max_{C \in \mathbb{C}(P_0)} \text{diam}(\text{co}(C)),$$

where $\mathbb{C}(P_0)$ denotes the collection of connected components of $\mathbb{G}_{\text{disk}}(r)$ at P_0 . The connectedness of each $C \in \mathbb{C}(P_0)$ implies that $\text{diam}(\text{co}(C)) \leq (n - 1)r$, and therefore, $\text{TC}(\mathbb{T}_{\text{ndzvs}}, \mathbb{CC}_{\text{rcmctr}}) \in O(n)$.

The lower bound in fact (i) is established by considering $P_0 \in \mathbb{R}^n$ such that $p^{[i+1]}(o) - p^{[i]}(o) = r$, $i \in \{1, \dots, n - 1\}$. For this configuration, we have $\text{diam}(\text{co}(P_0)) = (n - 1)r$ and, therefore, $\text{TC}(\mathbb{T}_{\text{ndzvs}}, \mathbb{CC}_{\text{rcmctr}}, P_0) \geq n - 1$.

Fact (ii). In the r -limited Delaunay graph, two agents on the line that are at most at a distance r from each other are neighbors if and only if there

are no other agents between them. Also, note that the r -limited Delaunay graph and the r -disk graph have the same connected components (cf., Theorem 2.8). An argument similar to the one used in the proof of fact (i) above guarantees that the connectivity constraints imposed by the constraint sets

$\mathcal{X}_{\text{disk}}(p^{[i]}, \{p_{\text{rcvd}} \mid \text{for all non-null } p_{\text{rcvd}} \in y^{[i]}\})$ are again superfluous.

Consider first the case when $G_{\text{LD}}(r)$ is connected at P_0 . Note that this is equivalent to $G_{\text{disk}}(r)$ being connected at P_0 . Without loss of generality, assume that the agents are ordered from left to right according to their identifier, that is, $p^{[1]}(0) \leq \dots \leq p^{[n]}(0)$. The evolution of the network under

$\text{CC}_{\text{cromentr}}$ can then be described as the discrete-time dynamical system

$$\begin{aligned} p^{[1]}(\ell + 1) &= \frac{1}{2}(p^{[1]}(\ell) + p^{[2]}(\ell)), \\ p^{[2]}(\ell + 1) &= \frac{1}{2}(p^{[1]}(\ell) + p^{[3]}(\ell)), \\ &\vdots \\ p^{[n-1]}(\ell + 1) &= \frac{1}{2}(p^{[n-2]}(\ell) + p^{[n]}(\ell)), \\ p^{[n]}(\ell + 1) &= \frac{1}{2}(p^{[n-1]}(\ell) + p^{[n]}(\ell)). \end{aligned}$$

Note that this evolution respects the ordering of the agents. Equivalently, we can write $P(\ell + 1) = AP(\ell)$, where $A \in \mathbb{R}^{n \times n}$ is the matrix given by

$$A = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & \dots & \dots & 0 \\ 0 & 1 & \dots & \dots & 0 & 0 \\ 1 & 0 & \frac{1}{2} & \frac{1}{2} & \dots & 0 \\ 0 & 1 & 0 & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \end{bmatrix}$$

... 1

$$\begin{bmatrix} 0 & \dots & \dots & 0 \\ & & & 2 \\ & & & 0 \\ & & & 2 \\ & & & 1 \\ & & & 1 \end{bmatrix}$$

Note that $A = A\text{Trid}_{n-2}$, 0, as defined in Section 1.6.4. Theorem 1.86(i)

implies that, for $P_{\text{ave}} = \frac{1}{n} \mathbf{1}^T P_0$, we have that $\lim_{\ell \rightarrow +\infty} P(\ell) = P_{\text{ave}} \mathbf{1}_n$. (Note that this is the maximum time required for $P(\ell) - P_{\text{ave}} \mathbf{1}_n$ (over all initial conditions in $\mathbb{R}^n$) is $\Theta(n^2)$.)

$P_{\text{ave}} \mathbf{1}_n$, and

implies that agents rendezvous at the location given by the average of their

initial positions. In other words, the asymptotic rendezvous position for this case can be expressed in closed form, as opposed to the case with the r -disk graph.)

Next, let us convert the contraction inequality on 2-norms into an appropriate inequality on ∞ -norms.

Note that $\text{diam}(\text{co}(P_0)) \leq (n-1)r$ because

$G_{LD}(r)$ is connected at P_0 .

Therefore,

$$\|P_0 - P_{ave}\|_\infty = \max_{i \in \{1, \dots, n\}} |p_i(0) - P_{ave}| \leq |p_1(0) - p_n(0)| \leq (n-1)r.$$

For ℓ of order $n^2 \log \eta^{-1}$, we use this bound on $\|P_0 - P_{ave}\|_\infty$ and the basic

inequalities $\|v\|_\infty \leq \|v\|_2 \leq \sqrt{n} \|v\|_\infty$ for all $v \in \mathbb{R}^n$, to obtain

$$\begin{aligned} \|P(\ell) - P_{ave}\|_\infty &\leq \|P(\ell) - P_{ave}\|_2 \leq \eta \|P_0 - P_{ave}\|_2 \\ &\leq \eta \sqrt{n} \|P_0 - P_{ave}\|_\infty \leq \eta n(n-1)r. \end{aligned}$$

This means that $(r\epsilon)$ -rendezvous is achieved for $\eta \sqrt{n(n-1)}r = r\epsilon$, that is,

in time $O(n^2 \log \eta^{-1}) = O(n^2 \log(n\epsilon^{-1}))$.

Next, we show the lower bound. Consider the unit-length eigenvector

$v_n = \frac{1}{\sqrt{2}} (\sin \frac{\pi}{n}, \dots, \sin \frac{n\pi}{n})^T \in \mathbb{R}^n$ of $\text{Trid}_{n-1}(\frac{1}{n}, 0, \frac{1}{n})$ corresponding to the largest singular value $\cos(\frac{\pi}{2n})$. For $\mu = \frac{1}{\sqrt{2}} rn^{5/2}$, we then define the initial condition

$$P \Sigma = \mu P \begin{pmatrix} 0 & \Sigma \\ 0 & v_{n-1} \end{pmatrix} \in \mathbb{R}^n.$$

One can show that $p^{[i]}(0) < p^{[i+1]}(0)$ for $i \in \{1, \dots, n-1\}$, that $P_{ave} = 0$, and that $\max\{p^{[i+1]}(0) - p^{[i]}(0) \mid i \in \{1, \dots, n-1\}\} \leq r$. Using Lemma 1.82

and because $\|w\|_\infty \leq \|w\|_2 \leq \sqrt{n} \|w\|_\infty$ for all $w \in \mathbb{R}^n$, we compute

$$\begin{aligned} \|P_0\|_\infty &= \frac{\|P_+\|_\infty}{\sqrt{2}} = \frac{\|P_+\|_2}{\sqrt{2}} \geq \frac{1}{\sqrt{2}} \frac{\|P_+\|_2}{\sqrt{2}} \\ &\geq \frac{rn}{10\sqrt{2}} \|v_{n-1}\|_2 = \frac{rn}{10\sqrt{2}}. \end{aligned}$$

The trajectory $P(\ell) = (\cos(\frac{\pi}{2n}))^\ell P_0$ therefore satisfies

Therefore, that $"P(\ell)"_\infty$ is larger than $\frac{1}{10} r \varepsilon$ so long as $\frac{1}{\sqrt{n}} \cos\left(\frac{1}{\sqrt{n}}\right) \geq \frac{1}{10}$ is, so long as

$$\ell < \frac{\log(\varepsilon^{-1}n) \log(5\sqrt{2})}{\log\left(\frac{1}{\sqrt{n}} \cos\left(\frac{1}{\sqrt{n}}\right)\right)}$$

In exercise [E4.7](#), the reader is asked to show that the asymptotics of this bound correspond to the lower bound in fact (i).

Now consider the case when $\mathcal{C}_{LD}(r)$ is not connected at P_0 . Note that the connected components do not change along the network evolution. Therefore, the previous reasoning can be applied to each connected component.

Since the number of agents in each connected component is strictly less than

n , the time complexity can only but improve. Therefore,

$$\text{we conclude that } \text{TC}(\mathcal{T}_{\text{rndzvs}}, \text{CC}_{\text{crcmctr}}) \in \Theta(n^2 \log(n\varepsilon^{-1})).$$

Fact (iii). Recall from the proof of Theorem 4.6.2 that the evolution under $\text{CC}_{\text{pll-crcmctr}}$ of the robotic network $\mathcal{S}_{\infty\text{-disk}}$ (in d dimensions) can be alternatively described as the evolution under $\text{CC}_{\text{crcmctr}}$ of d robotic networks $\mathcal{S}_{\text{disk}}$ in $\mathbb{R}$ (see Exercise E4.4). Fact (iii) now follows from fact (i).

□

4.6.4 Proof sketch of Theorem 4.20

Here, we only provide a sketch of the proof of Theorem 4.20. Fact (i) is a consequence of the control function definition of the nonconvex_crcmctr law in Section 4.3.4 and Lemma 4.11. Fact (ii) follows from the fact that the law $\text{CC}_{\text{nonconvex\_crcmctr}}$ (with control magnitude bounds) achieves the

ε -rendezvous task $\mathcal{T}_{\varepsilon\text{-rndzvs}}$ and fact (i).

To show that, on the network $\mathcal{S}_{\text{vis-disk}}$, the law $\text{CC}_{\text{nonconvex\_crcmctr}}$ (with control magnitude bounds) achieves the ε -rendezvous task $\mathcal{T}_{\varepsilon\text{-rndzvs}}$, one can follow the same overapproximation strategy that we used in the proof of Theorem 4.16, STEP 1.; that is,

(a) define a set-valued dynamical system $(Q_\delta^n, Q_\delta^n, T)$ such that the evo-

lutions of $(\mathcal{S}_{\text{vis-disk}}, \text{CC}_{\text{nonconvex\_crcmctr}})$ starting from an initial configuration where $\mathcal{C}_{\text{vis-disk}, Q_\delta}$ is connected are contained in the set of evolutions of the set-valued

dynamical system; and

(b) establish that any evolution of $(Q_\delta^n, Q_\delta^n, T)$ converges to a point

in $\text{diag}(Q^n)$ (note that the point might be different for different evolutions).

We refer to [Ganguli et al. \(2009\)](#) for a detailed development of this proof strategy. Here, we only remark that in order to carry out (b), the proof uses the LaSalle Invariance Principle in Theorem [1.21](#), with the perimeter of the relative convex hull of a set of points as Lyapunov function.

7. Discussion

1. -69 (1949)

Snap-Stabilizing Tasks in Anonymous Networks

Emmanuel Godard^(B)

Aix-Marseille Université, CNRS, Centrale Marseille, LIF, Marseille, France

emmanuel.godard@lif.univ-mrs.fr

Abstract. We consider snap-stabilizing algorithms in anonymous networks. Self-stabilizing algorithms are well known fault tolerant algorithms: a self-stabilizing algorithm will eventually recover from arbitrary transient faults. On the other hand, an algorithm is snap-stabilizing if it can withstand arbitrary initial values and immediately satisfy its safety requirement. It is a subset of self-stabilizing algorithms. Distributed tasks that are solvable with self-stabilizing algorithms in anonymous networks have already been characterized by Boldi and Vigna in [BV02b].

In this paper, we show how the more demanding snap-stabilizing algorithms can be handled with standard tools for (not stabilizing) algorithms in anonymous networks. We give a characterization of which tasks are solvable by snap-stabilizing algorithms in anonymous networks. We also present a snap-stabilizing version of Mazurkiewicz' enumeration algorithm.

This work exposes, from a task-equivalence point of view, the complete correspondence between self or snap-stabilizing tasks and distributed tasks with various termination detection requirements.

1 Introduction

In the world of fault-tolerance, distributed tasks that admits self-stabilizing solutions have been long studied [Doloo]. An algorithm is self-stabilizing if, starting from arbitrary initial values in the registers used by the algorithm, it can eventually stabilize to a

correct final value. In particular, when looking at some computed values, the algorithm can output incorrect values as long as it eventually outputs correct ones.

In contrast, an algorithm is snap-stabilizing if it can withstand arbitrary initial values and output only correct values [BDPV99]. Snap-stabilizing tasks form a subset of self-stabilizing tasks where the algorithm is required to retain computed values until it is “sure” that they are correct. Snap-stabilizing algorithms have really interesting properties, they can withstand arbitrary transient failures, while at the same time, improving on self-stabilizing algorithms about a key point: the stabilization moment is not unknown: when a response is given, it is correct.

We present here the first characterization of snap-stabilizing tasks on anonymous networks. Not only we are reusing techniques borrowed from the study of the non-stabilizing tasks in anonymous networks and show they apply also

 Springer International Publishing AG 2016

B. Bonakdarpour and F. Petit (Eds.): SSS 2016, LNCS 10083, pp. 170 – 184, 2016.

DOI: 10.1007/978-3-319-49259-9-14

here, but we complete the correspondence between self/snap-stabilizing tasks and termination detection.

How does snap-stabilizing tasks differs from self-stabilizing tasks has not been considered so far to the best of our knowledge. Here we show that, on anonymous networks, there are tasks that admits self-stabilizing solutions but that have no snap-stabilizing ones. We show that the difference between self and snap stabilization is actually the same one gets with non-stabilizing tasks when considering implicit vs explicit termination.

1.1 Our Result

We give the first characterization of the computability of snap-stabilization. In order to show that it complements known results about self-stabilizing and non self-stabilizing tasks in anonymous networks, we recall the previous equivalence established by Boldi and Vigna. Solving a task means solving a given specification linking inputs labels to output labels for a given set of graphs. Informally an algorithm has implicit termination if it is possible to write numerous time a (tentative) solution in the dedicated Out register. An algorithm has explicit termination when it is possible to write in Out only once. Whenever the Out register is defined, this means that (locally) the algorithm has terminated its computation.

Theorem 1 (Boldi and Vigna [BV01, BV02b]). *A task is solvable on a family of anonymous networks by a self-stabilizing algorithm if and only if it is solvable with implicit termination.*

The “only if” part being obvious, the merit of [BV02b] is to show that there is a universal algorithm to solve tasks (that are at all solvable) by a self-stabilizing algorithm on anonymous networks, and that the condition for solvability (informally speaking: stability of the specification by lifting) is exactly the one required by implicit termination. In other words, once

a problem is solvable with implicit termination, it admits a reliable self-stabilizing solution without any additional condition.

Theorem 2 (this paper). *A task is solvable on a family of anonymous networks by a snap-stabilizing algorithm if and only if it is solvable with explicit termination.*

As in the Boldi and Vigna result, the “only if” part is immediate. We therefore focus on establishing the “if” part. So the main contribution of this paper is a universal snap-stabilizing algorithm that solves the task at hand if this task satisfies the condition for being solvable by an algorithm with explicit termination. This condition was first given by [BV01].

In the presentation we will use here, it was also presented in [CGM08]. We first prove our results for terminating tasks in the asynchronous model, then we show how to extend the technique for long lived tasks in the synchronous model (for simplicity of exposition).

The presentation follow the order of [BV02b]. Instead of the view algorithm, we use Mazurkiewicz' algorithm. A variation of this algorithm was proved to be self-stabilizing in [God02], in Mazurkiewicz' model, a model that offers strong synchronization between neighbours. We present here a version for the state model.

1.2 Related Work

Snap-stabilizing algorithms were introduced in [BDPV99]. Since this definition is very general, we choose here to formally distinguish, as in the non-stabilizing case, snap-stabilization from self-stabilization: we consider only specification where the Out register has to be readily correct when defined. In [CDV09], a general transformation technique is given to obtain simple snap-stabilizing algorithms. In [CDV09], the authors expose a sufficient condition for snap-stabilizing algorithm equivalence. In this paper, we prove the task equivalence between snap-stabilization and explicit termination. The complexity of the respective solutions might not be equivalent.

In [AD14], a probabilistic correction condition is proposed for snap-stabilizing algorithms. This condition has the interest that it defines, in a sound way, what is a Las Vegas self-stabilizing algorithm.

On the distributed computability side, the first complete characterization of tasks that admits self-stabilizing algorithms has been given in [BV02b]. Here, we use a mix of different techniques, some of which were first introduced in [CGM08].

There is an unpublished version of Mazurkiewicz' algorithm in the communication model of this paper but without transient faults in [Chao6, chap. 4], where the model is coined the "cellular model". One of the main advantage of Mazurkiewicz' algorithm is that it is always stabilizing, contrary to the view algorithm of

[BV02b] where it is necessary to know or derive an estimate of the size to make it stabilizing.

2 *Definitions and Notations*

2.1 **Basic Definition for Computability**

A network is represented by a graph or digraph where vertices corresponds to nodes and edges or arcs corresponds to (possibly asymmetric) communication links. We consider a set of labels Λ . Labels are used to represent the local states of parts of the communication network.

So we consider labelled graphs in the general sense. Nodes can be labelled (internal state of the nodes), arcs can be labelled (messages in transit, port numbering). We will use $\mathbf{G}$ to denote a (di)graph with all its associated labels. Since the input labels can be encoded in the labels, we consider all labelled graphs as the possible inputs for distributed algorithms. The set of all labelled graphs is denoted G .

Given a network G and a vertex v in G , we assume that the state of v during the execution of any algorithm is of the form $(\text{Mem}(v), \text{Out}(v))$: $\text{Mem}(v)$ is the memory of v , $\text{Out}(v)$ will contain its output value, i.e. the result of the computation at v . When Out is not defined, it contains the value .

A distributed algorithm is an algorithm that is replicated on every node and operates on the local state of the node v by way of communication with the neighbours of v . The communication here is done in the *shared variables* model, that is also called the *state* model. A distributed algorithm is a set of rules (a pair of precondition and command) that describe how a processor has to change its current state (the command) according to the state of *all its neighbors* (the precondition or guard). We say that a rule R is activable at a processor v if the neighborhood of v satisfies the precondition of R . In this case, the vertex v is also said to be activable. If a rule R is activable in v , an atomic move for v consists of reading the states of all its neighbors, computing a new value of its state according to the command of R , and writing this value to the local memory Mem . An execution is a sequence of such moves. If more than one rule is activable at a node, one is chosen non-deterministically. Of course, it is possible to have priorities for rules, and to discard this non-determinism. If only one such move can occur at a time, this is a *central daemon* execution. If any combination of such moves can occur, this is an *asynchronous daemon* execution. If all possible moves occur, this is the *synchronous daemon* execution. We consider here the asynchronous daemon (which contains the synchronous daemon).

A vertex-relabelling relation is a relation between labelled graphs where the underlying graphs are

identical. The evolution of the global system can be seen as a sequence of relabelling steps where only local part of the labels of the graphs are modified, according to rules prescribe by the algorithm and the kind of daemon that is considered. Under a given execution ρ , the global configuration of the system is described by the sequence of labelled graphs $(G, \text{Mem}_1), (G, \text{Mem}_2), \dots$; this is abbreviated in $(G_1, G_2, \dots)$ for convenience. Such a sequence is an execution of algorithm Algo . If the sequence is finite, that is if there is a step where no rule is applicable, we say that the execution has terminated and denote by G_f the labelled graph with component Out, $G_f = (G, \text{Out})$, it is the *terminal state* of the execution.

We formally define now what is a terminating distributed problem.

Definition 3. A terminating distributed task is a couple (F, S) where F is a family of labelled graphs and S is a vertex-relabelling relation.

The *specification* S is a general way to describe our distributed problem in terms of relation between inputs and outputs. This description is independent of the domain F where we want to solve our problem.

Definition 4. Given $G \in F$, an algorithm Algo solves S on G if for any execution:

stabilization there is a terminal state G_f ;

decision Out_v is written exactly once for any vertex v ;

correction $G \leq G_f$.

Definition 5. The task (S, F) is solvable if there exists an algorithm Algo

such that Algo solves S for all $\mathbf{G} \in F$.

When, besides **correction**, the **stabilization** property is the only property, we talk about *implicit termination* (or message termination [Tel00]). When we have both **stabilization** and **decision**, we talk about *explicit termination*. Those are the main termination mode that are classically considered in distributed algorithms. See also [CGM08, GMT10] for other types of termination.

2.2 Self- and Snap-Stabilization

Informally, a distributed algorithm is said to be self-stabilizing if an execution starting from any arbitrary global state has a suffix belonging to the set of legitimate states. A distributed algorithm is snap-stabilizing if an execution starting from any arbitrary global state has *all its causal suffixes* belonging to the set of legitimate states. Note that when we consider terminating tasks, the set of legitimate states corresponds simply to the set of S admissible outputs for the given input graph, that is the set $\mathbf{G}^j \mathbf{G}^i \mathbf{G}^j$.

In anonymous networks, self-stabilization, on the task level, relates to implicit termination whereas snap-stabilization relates to explicit termination. To emphasize this relation, we first define snap-stabilization for terminating tasks. Then we will proceed to long lived ones.

Given the registers can be corrupted in any way, there would be no way to compute snap-stabilizing task if there weren't a restriction on the outputs to be considered. Formally, the output values are to be correct only when they have been causally influenced by a special “request” event. This event is an event exterior to the algorithm and occurs *after* the end of the faults that led to arbitrary incorrect values.

The setting is the following, given an arbitrary state of the system, one or more external actions, *the requests*, are applied at some nodes $U \subseteq V$. Then the system

evolves according to the daemon and the algorithm: at one step, some nodes (or only one in the central daemon case) are activable and activated (their actions are processed). Given a sequence $A_1, A_2, \dots$ of activated nodes, an execution ρ is the sequence $\mathbf{G} = \mathbf{G}_0, \mathbf{G}_1, \mathbf{G}_2, \dots$ of labelled obtained by applying to $\mathbf{G}_{i-1}$ the actions for the nodes of A_i .

A node v is *causally influenced* by U if there exists a path $u_0, u_1, \dots, u_k$

such that $u_0 \in U$, and there exists a strictly increasing function $\sigma : \mathbb{N} \rightarrow \mathbb{N}$

$\forall i \geq 1, u_i \in A_{\sigma(i)}$ and $u_k = v$. Given $i \in \mathbb{N}$, we define V_i to be the set of vertices that have been causally influenced at step i , that is if $\sigma(k) \leq i$. Given a set U of requested processes, $C(U)$ is the set of causally influenced processes by processes of U . We define $C_i(\mathbf{G}) = \mathbf{G}|_{V_i}$ to be the subgraph of $\mathbf{G}_i$ that is induced by V_i .

Definition 6. An algorithm *Algo* is snap-stabilizing to the terminating task

(S, F) on $\mathbf{G} \in F$ if, starting from any global configuration,

stabilization *there is a terminal state tt_f ;*

causal decision Out_v *is written exactly once when v has been causally influenced;*

S-correction $ttStt_f$.

Synchronous Daemon for Long Lived Tasks.

The specification of a long lived tasks is given by a set of infinite sequences $\mathcal{S} = (S_1, S_2, \dots)$, where the S_i are relabelling relations as in the terminating case. Given a set P of requested processes, $C_{\text{sync}}(P)$ is the set of causally influenced processed by processes of P .

Definition 7. *Given a set U of requested processes, an algorithm Algo snap-stabilizes to S against the synchronous daemon if $\mathbf{G}_{\text{Out}}|_{C_{\text{sync}}(P)} \in S$.*

3 Terminating Tasks

We start by considering snap-stabilizing terminating tasks. This is where it is easier to show how the general techniques from explicitly terminating non-stabilizing tasks can be extended to the snap-stabilizing case as well.

3.1 Digraphs and Fibrations

Defnitions. In the following, we give the definitions for the tools introduced by Boldi and Vigna, and extensively studied in [BVo2a], to characterize self-stabilizing tasks in [BVo2b]. To introduce the main tool, that is *fibrations*, we need to consider directed graphs (or digraphs) with multiple arcs and self-loops. A *digraph* $D = (V(D), A(D))$ is defined by a set $V(D)$ of vertices and a set $A(D)$ of arcs. Given an arc a , we say that a is adjacent to the vertices $s(a)$ and $t(a)$, the source and target of the arc. An undirected graph tt corresponds to the digraph $\text{Dir}(tt)$ obtained by

replacing all edges of tt by the two corresponding arcs. In the following, we will not distinguish tt and $Dir(tt)$ when the context permits. The family of all digraphs with multiple arcs and self-loops is denoted $\mathcal{D}$.

A dipath π of length p from u to v in D is a sequence of arcs $a_1, a_2, \dots, a_p$ such that $s(a_1) = u$, $t(a_p) = v$ and for all i , $s(a_{i+1}) = t(a_i)$. A digraph is strongly connected if there is a path between all pairs of vertices. We assume all digraphs

to be strongly connected. Note that D is embedded in dir via dir . Moreover we only consider undirected networks, i.e. symmetric digraphs, however, the reader shall note that they can be fibrations of general digraphs. Labelled digraphs will be designated by bold letters like **D**, **G**, **H**...

A homomorphism γ between the digraphs D and D' is a mapping $\gamma: V(D) \cup A(D) \rightarrow V(D') \cup A(D')$ such that for each $a \in A(D)$, $\gamma(s(a)) = s(\gamma(a))$ and $\gamma(t(a)) = t(\gamma(a))$. A homomorphism $\gamma: V(D) \cup A(D) \rightarrow V(D') \cup A(D')$ is an isomorphism if γ is bijective. Homomorphisms will also preserve labelling.

In a digraph **G**, given $u_0 \in V(\mathbf{G})$ and $r \in \mathbb{N}$, we denote by $B_{\mathbf{G}}^-(u_0, r)$ the in-ball of center u_0 and radius r , that is the set of vertices v and arcs a such that there is a dipath of length at most r from v to u_0 .

Fibrations and Quasi-Fibrations. The notion of fibrations have been proved fundamental for the study of distributed tasks in anonymous networks in the work of Boldi and Vigna [BV02b], definitions and main properties are presented in [BV02a]. This notion enables to express the “similarity” between two anonymous networks in the state model.

A digraph $\mathbf{D}$ is a fibration of a digraph $\mathbf{D}'$ via φ if φ is a homomorphism

from $\mathbf{D}$ to $\mathbf{D}'$ such that each arc $\alpha \in A(\mathbf{D})$ and for each vertex $v \in V(\mathbf{D})$ (resp. $v \in V(\mathbf{D}')$), there exists a unique arc $\alpha' \in A(\mathbf{D}')$ such that $t(\alpha) = v$ (resp. $s(\alpha') = v$).

The following lemma shows the importance of fibrations when we deal with anonymous networks. This is the counterpart of the lifting lemma that Angluin gives for coverings of simple graphs [Ang80] and the proof can be found in [BCG+96, BV02b, CM07].

Lemma 8 (Lifting Lemma [BCG+96]). *If $\mathbf{D}$ is a fibration of $\mathbf{D}'$ via φ , then any execution of an algorithm Algo on $\mathbf{D}$ can be lifted up to an execution on $\mathbf{D}'$, such that at the end of the execution, for any $v \in V(\mathbf{D})$, v is in the same state (Mem, Out) as $\varphi(v)$.*

In the following, one also needs to express similarity between two digraphs up to a certain distance. The notion of quasi-coverings was introduced as a formal tool in [MMW97, GM02] for this purpose. The next definition is an adaptation of this tool to fibrations.

Definition 9. *Given digraphs $\mathbf{K}$ and $\mathbf{H}$, and integer r and $v \in V(\mathbf{K})$ and an homomorphism γ from $B_{\mathbf{K}}^-(v, r)$ to $\mathbf{H}$, $\mathbf{K}$ is a quasi-fibration of $\mathbf{H}$ of center v and radius r via γ if there exists a finite or infinite digraph $\mathbf{G}$ such that $\mathbf{G}$ is a fibration of $\mathbf{H}$ via a homomorphism φ and there exists $w \in V(\mathbf{G})$ and an isomorphism δ from $B_{\mathbf{K}}^-(v, r)$ to $B_{\mathbf{G}}^-(w, r)$ such that for any $x \in V(B_{\mathbf{K}}^-(v, r)) \in \cup A(B_{\mathbf{K}}^-(v, r))$, $\gamma(x) = \varphi(\delta(x))$*

If a digraph $\mathbf{G}$ is a fibration of $\mathbf{H}$, then for any $v \in V(\mathbf{G})$ and for any $r \in \mathbb{N}$, $\mathbf{G}$ is a quasi-fibration of $\mathbf{H}$, of

center v and of radius r . Reversely, if $\mathbf{K}$ is a quasi-fibration of $\mathbf{H}$ of radius r strictly greater than the diameter of $\mathbf{K}$, then $\mathbf{K}$ is a fibration of $\mathbf{H}$. The following lemma is the counterpart of the lifting lemma for quasi-fibrations.

Lemma 10 (Quasi-Lifting Lemma). *Consider a digraph $\mathbf{K}$ that is a quasi-fibration of $\mathbf{H}$ of center v and of radius r via γ . For any algorithm Algo, after r rounds of the synchronous execution of an algorithm Algo on $\mathbf{K}$, v is in the same state as $\gamma(v)$ after r rounds of the synchronous execution of Algo on $\mathbf{H}$.*

3.2 Necessary Condition

It is obvious that the necessary condition of [CGMo8] applies here, as well as its proof. We present the result briefly to make the paper self-contained.

We denote $D_{\bullet}$ the disjoint union $\{(\mathbf{G}, v) \mid \mathbf{G} \in D, v \in V(\mathbf{G})\}$. Given a family $F \subset G$, we denote by $F_{\bullet}$ the set $\{(\mathbf{G}, v) \mid \mathbf{G} \in F, v \in V(\mathbf{G})\}$. A function $f : D_{\bullet} \rightarrow \cup\{\perp\}$ is an *output function* for a task $(S, \mathcal{F})$ if for each network $\mathbf{G}$ the labelling obtained by applying f on each node $v \in V(\mathbf{G})$ satisfies the specification S , that is $\mathbf{G}S(\mathbf{G}, \lambda)$ where $\lambda(v) = f(\mathbf{G}, v)$.

In order to give our characterization, we need to formalize the following idea. When the neighbourhood at distance k of two processes v_1, v_2 in two digraphs $\mathbf{D}_1, \mathbf{D}_2$ cannot be distinguished (this is captured by the notion of quasi-fibrations and Lemma 10), and if v_1 computes its final value in k rounds, then v_2 computes the same final value in the same number of rounds.

Definition 11. Given a function $r : D_{\bullet} \rightarrow \mathbb{N} \cup \{\infty\}$ and a function $f : D_{\bullet} \rightarrow \Lambda$, the function f is *r -lifting closed* if for all $\mathbf{K}, \mathbf{H} \in D$ such that $\mathbf{K}$ is a quasi-fibration of $\mathbf{H}$, of center $v \in V(\mathbf{G})$ and of radius k via γ with $k \geq \min\{r(\mathbf{G}, v), r(\mathbf{H}, \gamma(v))\}$, then $f(\mathbf{G}, v) = f(\mathbf{H}, \gamma(v))$.

In the previous definition, the value of $r(\mathbf{G}, v)$ can be understood as the number of rounds needed by v to compute in the synchronous execution its final value in $\mathbf{G}$.

We give now the characterization of terminating snap-stabilizing tasks. We give the proof of the necessary condition. The converse will be proved in the following section, by introducing a snap-stabilizing version of Mazurkiewicz' algorithm.

Theorem 12. Let $t \in \mathbb{N} \cup \{\infty\}$. A terminating task $(S, \mathcal{F})$ is computable by a terminating snap-stabilizing algorithm if and only if there exists a function $r : D_{\bullet} \rightarrow \mathbb{N} \cup \{\infty\}$ and a function $f : D_{\bullet} \rightarrow \Lambda \cup \{\perp\}$ such that,

- 12.i for all $(\mathbf{G}, v) \in D_{\bullet}$, $r(\mathbf{G}, v) = \infty$ if and only if $f(\mathbf{G}, v) = \perp$;
- 12.ii f and r are r -lifting-closed;

Proof (of the necessary condition). Consider Algo a distributed algorithm that snap-stabilizes to S on in t rounds.

We construct r and f by considering a subset of the possible executions of Algo . We consider the synchronous execution of Algo on any digraph $\mathbf{G}$ with initial values *init*. For any $v \in V(\mathbf{G})$, if $\perp$

$\text{Out}(v) =$ during the whole execution, then we set $f(\mathbf{G}, v) =$ and $r(\mathbf{G}, v) =$. This is possible since it could be that ζ . Let r_v be the first causal round after which $\text{Out}(v) =$; in this case, if $r_v \leq t$, we set $f(\mathbf{G}, v) = \text{Out}(v)$ and $r(\mathbf{G}, v) = r_v$. If $t < r_v$, then we set $f(\mathbf{G}, v) =$ and $r(\mathbf{G}, v) =$. By construction 12.i is satisfied.

We also show that f is an output function and that f and r satisfy 12.ii. Consider two digraphs $\mathbf{K}$ and $\mathbf{H}$ such that $\mathbf{K}$ is a quasi-fibration of $\mathbf{H}$, of center $v_0 \in V(\mathbf{K})$ and of radius k via γ with $k \leq r_0 = \min \{r(\mathbf{G}, v_0), r(\mathbf{H}, \gamma(v_0))\}$. If $\zeta \leq r_0$, then $r(\mathbf{K}, v_0) = f(\mathbf{H}, \gamma(v_0))$ and $f(\mathbf{G}, v_0) = f(\mathbf{H}, \gamma(v_0)) =$. $\perp$

Otherwise, from Lemma 10 (the initial labelling *init* is liftable), we know that after r_0 rounds, $\text{Out}(v_0) = \text{Out}(\gamma(v_0))$. Thus $r_0 = r(\mathbf{G}, v_0) = r(\mathbf{H}, \gamma(v_0))$ and $f(\mathbf{G}, v_0) = f(\mathbf{H}, \gamma(v_0))$. Consequently, f and r are r -lifting closed.

4 Main Algorithm

In this section, we present a general algorithm for which we use as parameter elements from the terminating task (S, τ) we are interested in, in order to obtain our sufficient condition. This algorithm is a combination of a snap-stabilizing enumeration algorithm, adapted from [God02] and a generalization of an algorithm of Szymanski, Shy and Prywes (the SSP algorithm for short) [SSP85].

The algorithm in [God02] is described in a different model, where each computation step involves some strong synchronization between adjacent processes. It is a self-stabilizing adaptation of an enumeration algorithm presented by Mazurkiewicz' in [Maz88]. The SSP algorithm enables to detect the global termination of an algorithm provided the processes know a bound on the diameter of the graph. The Mazurkiewicz-like algorithm always stabilizes on any network G and during its execution, each process v can reconstruct at some computation step i a digraph $G_i(v)$ such that G is a quasi-fibration of $G_i(v)$. However, this algorithm does not enable v to compute effectively the radius of this quasi-fibration. We use a generalization of the SSP algorithm to compute a lower bound on this radius, as it has already been done in Mazurkiewicz's model [GMT10] and in the message passing model [CGMo8].

4.1 Modifying Mazurkiewicz' Enumeration Algorithm

An enumeration algorithm on a network G is a distributed algorithm such that the result of any

computation is a labelling of the vertices that is a bijection from $V(\mathbf{G})$ to $1, 2, \dots, V(\mathbf{G})$.

We give first a general description of the algorithm. Every vertex attempts to get its own name (which shall be an integer between 1 and $V(\mathbf{G})$), however here we would need more work to enforce this, however since this is not need for our purpose, these technicalities will be skipped, see [God02]). A vertex chooses a name and broadcasts it together with its neighborhood all over the network. If a vertex u discovers the existence of another vertex v with the same name, then it compares its *local view*, i.e., the labelled in-ball of center

u , with the *local view* of its rival v . If the local view of v is “stronger”, then u chooses another

name. Node u also chooses another name if its appears twice in the view of some other vertex as a result of a corrupted initial state. Each new name is broadcast again over the network. At the end of the computation it is not guaranteed that every node has a unique name, unless the graph is fibration minimal. However, all nodes with the same name will have the same local view, i.e., isomorphic labelled neighborhoods.

The crucial property of the algorithm is based on a total order on local views such that the “strength” of the local view of any vertex cannot decrease during the computation. To describe the local view we use the following notation: if

v has degree d and its in-neighbors have names $n_1, \dots, n_d$, with $n_1, \dots, n_d$, then $N(v)$, the local view, is the d tuple $(n_1, \dots, n_d)$. Let T be the set of such ordered tuples. The alphabetic order defines a total order, $\prec$, on T . Vertices v are labelled by triples of the form (n, N, M) representing during the computation:

- $\underline{n}(v) \in N$ is the name of the vertex v ,
- $N(v) \in T$ is the latest view of v ,
- $M(v) \subset N \times T$ is the mailbox of v and contains all information received at this step of the computation.

We introduce other notations. We want to count the number of times a given name appear in a local view. For a local view N , and $n \in N$, we define $\delta_N(n)$ to be the cardinality of n in the tuple N . For a given view N , we denote by $sub(N, n, n^j)$ the copy of N where any occurrence of n is replaced by n^j .

The algorithm is given in Fig. 1. The rules are given in the priority order.

The labeling function obtained at the end of a run ρ of Mazurkiewicz' algorithm is noted π_ρ . If v is a vertex of G , the couple $\pi_\rho(v)$ associated with v is denoted $(n_\rho(v), M_\rho(v))$. We also note the final local view of v by $N_\rho(v)$. For a given mailbox M and a given $n \in N$, we note $Strong_M(n)$ the local view

Enum1 : Initialization
Enum2 : Diffusion rule

Guard :

• $Request(v_0)$

• There exists $v \in B(v_0)$ such that

Action :

• $\overline{M}(v_0) \neq M(v_0)$.

or $\overline{N}(v_0) \neq N(v_0)$ and $\overline{M}(v_0) \neq M(v_0)$,

or $\overline{M}(v_0) \neq M(v_0)$.

Action :

• $\overline{a}(v_0) := -1$.

• $\overline{M}(v_0) := \bigcup_{w \in B(v_0)} M(w) \cup$

$\{(n(v_0), N(v_0))\}$.

• $\overline{N}(v_0) := N(v_0)$.

• $a(v_0) := -1$.

Action :

• $n(v_0) = 1 + \max\{n \in N \mid (l, n, N) \in M(v_0) \text{ for some } l, N\}$.

• $M(v_0) = M(v_0) \cup \{(n(w), N(w)) \mid w \in B(v_0)\}$,

• $a(v_0) = -1$.

gSSPfix : Fix gSSP counter

Guard :

• If there exists $v \in B(v_0)$, $|a(v) - a(v_0)| \geq 2$ or $(M(v) \neq M(v_0) \text{ and } a(v) \neq -1)$

Action :

• $a(v_0) := -1$.

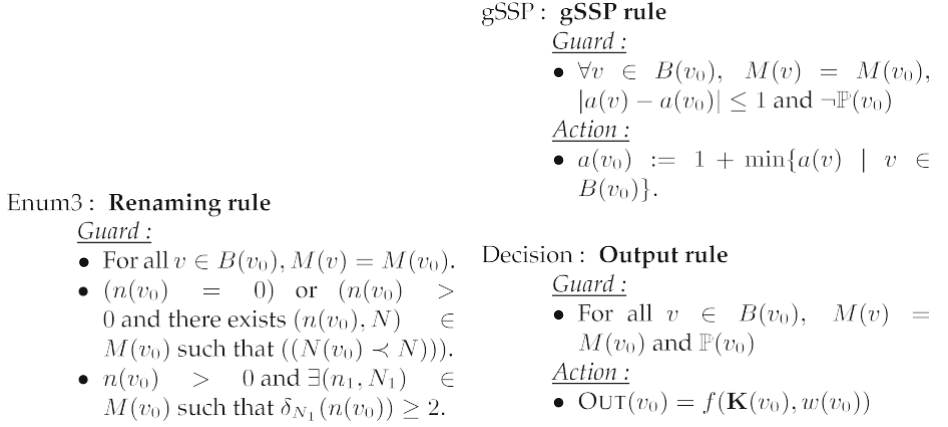


Fig. 1. Snap-stabilizing Mazurkiewicz' algorithm M_P , the parameter is the predicate P

that dominates all $N, (n, N) M$ (i.e. $N \text{ Strong}_M(n)$). Except for the first corrupted stages, $\text{Strong}_{M(v)}(n)$ is actually the “strongest local” view of n .

Theorem 13. *A run ρ of Mazurkiewicz’ Enumeration Algorithm on $\mathbf{G}$ with any initial values finishes and computes a final labeling π_ρ verifying the following conditions for all vertices v, v^j of $V(\mathbf{G})$:*

- 13.i $M_\rho(v) = M_\rho(v^j)$.
- 13.ii $\text{Strong}_{M_\rho(v)}(n_\rho(v)) = N(v) = N_\rho(v)$.
- 13.iii $n_\rho(v) = n_\rho(v^j)$ if and only if $N_\rho(v) = N_\rho(v^j)$.

Proof. This is similar to the one in [God02].

First we explain how it is possible to extract the map of the minimum base. This is usually done by considering the graphs induced by the numbers and associated local views that have maximal views. However, here, due to the arbitrary initial failures, the mailbox should be cleaned up before use. It is possible to have some maximal (n, N) but n does not actually exists on any v .

Now, each vertex shall compute locally the set of actual final names from the final mailbox M_ρ . We note $\mathbf{G}_\rho$ the graph defined by

$$V_\rho = \{n_\rho(v) \mid v \in V(\mathbf{G})\},$$

$$A_\rho = \{(n_\rho(v_1), n_\rho(v_2)) \mid (v_1, v_2) \in A(tt)\}.$$

For a mailbox M and an integer n , we define the set $V^M(n)$ by induction.

$$V_0^M = \{n\},$$

$$V_{i+1}^M = V_i^M \cup \{t \mid \exists s \, \mathcal{A}^M, \delta_{\text{Strong } M}(t) = 1\}.$$

If i_0 is such that $V^{M^*} = V^{M^*}$ then we define $V^M(n) = V^{M^*}$. Finally, we have,

$$i_0 \qquad i_0+1 \qquad i_0$$

Lemma 14 ([God02]). *For all $v \in V(\mathbf{G})$, $V^{M_\rho}(n_\rho(v)) = V_\rho$.*

By defining A^M by $\{(n_1, n_2) \mid n_1, n_2 \in V^M(n) \text{ and } \delta_{\text{strong}}(n_1)(n_2) = 1\}$, we

obtain a graph $\mathbf{G}_{M(v)} = (V^{M(v)}, A^{M(v)})$. We can not readily use $\mathbf{G}_{M(v)}$ since it could be that it is not in $\mathcal{G}$. We denote by $\mathbf{K}(v)$ the digraph that is in $\mathcal{G}$ and that is a quasi-fibration of $\mathbf{G}_{M(v)}$ of radius $\alpha(v)$ and of center $w(v)$. This can be found by enumerating (locally) all graphs and vertices of $\mathcal{G}$ until one is found. This semi-algorithm will terminate because of the following property.

Proposition 15. *Let P be the set of requesting processes. Let v that has been causally influenced by P , and such that $\alpha(v) > 0$. The graph $\mathbf{G}$ is a quasi-fibration of $\mathbf{G}_{M(v)}$ of center v and radius $\alpha(v)$.*

Proof. We add that every $w \in B(v, \alpha(v))$ has been influenced to the statement and prove this new statement by induction on i , the number of rounds since P has received the requests.

At round 0, no request has been yet processed by Enum1, i.e. the set of influenced nodes is empty and the property holds trivially.

Assume the property holds at round i and consider v_0 a vertex that is activated at round $i + 1$. We have to consider two cases, either v_0 was already influenced at round i or it is a newly influenced node.

If v_0 is a newly influenced node. The only rule of interest is gSSP because other rules are setting $a(v_0)$ to 1. But we show that v_0 cannot apply this rule. Indeed, assume $M(v_0) = \perp$, then, the causality path to v_0 starts in a root whose variables have been reset, and from which the causality chain of applications will propagate its new name. So $M(v_0)$ has to be updated to, at least, this name before being able to apply gSSP.

If v_0 has already been influenced then the induction statement applies at the previous round. Denote $a(v_0)$ the value of the counter at the end of round i and assume that for all $v \in N(v_0)$, $a(v) = a(v_0)$. We prove that the statement holds for $a(v_0) + 1$ at round $i + 1$.

If $a(v_0) = 0$ then, by the same argument as in the previous case, the neighbours of v_0 have all been influenced and the statement holds with a radius 1.

If $a(v_0) > 0$ then the neighbours have been influenced by induction assumption. Moreover, every $v \in N(v_0)$ is the center of a quasi-fibration of radius $a(v_0)$. Therefore, v_0 is the center of a quasi-fibration of radius $a(v_0) + 1$. Similarly, every $w \in B(v, a(v_0))$ has been influenced and the ball $B(v_0, a(v_0) + 1)$ is totally influenced. The statement holds at round $i + 1$.

The algorithm from Fig. 1 uses a special parameter P . This predicate will be formally defined below. It needs to make the counter a increase when what can be extracted from the mailboxes (that is the minimum base of $\mathbf{G}$) is the same locally. But it must also make the algorithm stop when there is enough information to conclude. This information is enough when the value r for the reconstructed base match the counter

of stability α .

Theorem 16. *With $P(v) := (\alpha(v) < r(K(v), n(v)))$, the algorithm $\mathcal{P}_{\text{snap}}^M$ stabilizes to S .*

Proof. Consider a node v just after it has applied rule Decision, we have $\text{Strong}(M(v))$ that is stable, $r(K(v), n(v)) \leq \alpha(v)$ and $\text{out}(v) = f(K(v), w(v))$. Since, by construction, $K(v)$ is a quasi-fibration of $G_{M(v)}$ of radius $\alpha(v)$ $r(K(v), n(v))$ and of center $n(v)$, and since f and r are r -lifting closed, $\text{Out}(v) = f(K(v), w(v)) = f(G_{M(v)}, n(v))$, and $r(K(v), w(v)) =$ —

$r(G_{M(v)}, n(v))$. From Proposition 15, since $\alpha(v) \geq r(G_{M(v)}, n(v))$ and since f

is r -lifting closed, $\text{Out}(v) = f(G_M(v), n(v)) = f(G, v)$.

5 Long Lived Tasks

From the results of the previous section, it is quite easy to derive a characterization for long lived tasks, that is tasks that do not terminate but evolve in a prescribe way. We mainly have to extend our previous techniques to sequences of labelled graphs. This can be done using the self-stabilizing clock uses in [BV02b]. For lack of space, we omit the proof and refer the interested reader to [BV02b].

Theorem 17. *A task (S, F) admits a snap-stabilizing synchronous solution if and only if there exists a function $r : F \rightarrow \mathbb{N} \cup \{\infty\}$ and a sequence of output functions $(f_1, f_2, \dots)$, $\forall i, f_i : F \rightarrow L \cup \{\perp\}$.*

- 17.i $\forall \mathbf{G} \in F, (f_1(\mathbf{G}), f_2(\mathbf{G}), \dots) \in S$,
- 17.ii r is r -lifting-closed;
- 17.iii $\forall i \in \mathbb{N}, f_i$ is r -lifting-closed.

6 Application

In this section, we show how the previous characterizations can be applied to characterize the families of homonymous rings where it is possible to solve the Mutual Exclusion problem. This is a long lived task that is defined by:

Safety: At every moment, there is at most one node with
Out = CriticalSection.

Liveness: There is no infinite suffix with requesting nodes
 and no nodes with
Out = CriticalSection.

As in the Election case in [CGM12, Sect. 6.4], we consider $\Lambda = \mathbb{N}_R$ and the set of homonymous rings R_Λ . Let $k \in \mathbb{N}$ and $\mathbf{G} \in R_\Lambda$, we denote $\mu_{\mathbf{G}}(k)$ the number of vertices with initial label equal to k in $\mathbf{G}$. More precisely, we consider the set of homonymous rings R_*

with a least one distinguished node, that is if $\mathbf{G} \in R_*$ then there is at least one integer k such that $\mu_{\mathbf{G}}(k) = 1$. All graphs of R_* are fibrations minimal, i.e. it is possible to solve Mutual Exclusion on any graph of R_* . Indeed, for a given $\mathbf{G}$, it is possible to design a sequence of output functions that simulate a token circulation by using the node u with unique name as leader. It is then possible to have a self-stabilizing solution to Mutual Exclusion on R_* (see [BY02b]).

However, it is not possible to solve the Mutual Exclusion

problem with a snap-stabilizing algorithm on R_* . Indeed, for any $n \in \mathbb{N}$, there exists a quasi-fibration $\mathbf{K} \in R_*$ of radius n via γ such that u has more than one pre-image. This remark prohibits the existence of r -lifting closed functions on R_* satisfying the unicity implied by the Safety condition. The subfamilies $\mathcal{F}_n^*$ that admit a snap-stabilizing solution to the mutual exclusion problem are therefore the families for which the radius of quasi-fibrations is bounded. This bound can be a bound on the size of the ring network, or, more generally, on the number of repetitions that a “name patterns” (a sequence of consecutive names on the ring) can have in F .

7 Conclusion

We have shown that for anonymous networks, the terminating tasks that can be solved by a snap-stabilizing algorithms are exactly the ones that can be solved by a distributed algorithm with explicit termination. This complements the already known task-equivalence between self-stabilizing terminating tasks

and distributed tasks computed with implicit termination. The important consequence is that the partial knowledge (like bound on the size, diameter etc.

.....
)

that could be used to get explicit termination in the non-stabilizing case are also the ones that can be used to have snap-stabilizing solutions.

For lack of space, we do not discuss probabilistic snap-stabilization [AD14]. It is not difficult to see that the techniques presented here enable to prove that a task has a probabilistic snap-stabilizing solution if and only it has a (non- stabilizing) Las Vegas solution.

An interesting open question, as in the self-stabilizing case, would be to find a *direct* way to transform any given anonymous algorithm into a snap-stabilizing one. Such transformation might have benefits regarding the complexity.

The author wishes to thank Jérémie Chalopin for sharing ideas and fruitful discussions about distributed computability in various settings, including some closely related to this paper.

References

[AD14] Altisen, K., Devismes, S.: On probabilistic snap-stabilization. In: Chat-terjee, M., Cao, J., Kothapalli, K., Rajsbaum, S. (eds.) ICDCN 2014. LNCS, vol. 8314, pp. 272–286. Springer, Heidelberg (2014). doi:[10.1007/978-3-642-45249-9_18](https://doi.org/10.1007/978-3-642-45249-9_18)

[Ang80] Angluin, D.: Local and global properties in networks of processors. In: Proceedings of the 12th Symposium on Theory of Computing, pp. 82–93 (1980)

[BCG+96] Boldi, P., Codenotti, B., Gemmell, P., Shammah, S., Simon, J., Vigna, S.: Symmetry breaking in anonymous networks: Characterizations. In: Proceedings 4th Israeli Symposium on Theory of Computing and Systems, pp. 16–26. IEEE Press (1996)

- [BDPV99] Bui, A., Datta, A.K., Petit, F., Villain, V.: State-optimal snap-stabilizing pif in tree networks. In *Workshop on Self-stabilizing Systems, ICDCS1999*, pp. 78–85. IEEE Computer Society (1999)
- [BV01] Boldi, P., Vigna, S.: An effective characterization of computability in anonymous networks. In: Welch, J. (ed.) *DISC 2001*. LNCS, vol. 2180, pp. 33–47. Springer, Heidelberg (2001). doi:[10.1007/3-540-45414-4_3](https://doi.org/10.1007/3-540-45414-4_3)
- [BV02a] Boldi, P., Vigna, S.: Fibrations of graphs. *Discrete Math.* **243**(243), 21–66 (2002)
- [BV02b] Boldi, P., Vigna, S.: Universal dynamic synchronous self-stabilization. *Distr. Comput.* **15**, 137–153 (2002)
- [CDV09] Cournier, A., Devismes, S., Villain, V.: Light enabling snap-stabilization of fundamental protocols. *ACM Trans. Auton. Adapt. Syst.* **4**(1), 6:1–6:27 (2009)
- [CGM08] Chalopin, J., Godard, E., Métivier, Y.: Local terminations and distributed computability in anonymous networks. In: Taubenfeld, G. (ed.) *DISC 2008*. LNCS, vol. 5218, pp. 47–62. Springer, Heidelberg (2008). doi:[10.1007/978-3-540-87779-0_4](https://doi.org/10.1007/978-3-540-87779-0_4)
- [CGM12] Chalopin, J., Godard, E., Métivier, Y.: Election in partially anonymous networks with arbitrary knowledge in message passing systems. *Distrib. Comput.* **25**(4), 297–311 (2012)

- [Cha06] Chalopin, J.: Algorithmique distribuée, calculs locaux et homomorphismes de graphes. PhD Thesis, Université de Bordeaux I (2006)
- [CM07] Chalopin, J., Métivier, Y.: An efficient message passing election algorithm based on mazurkiewicz's algorithm. *Fundam. Inform.* **80**(1–3), 221–246 (2007)
- [Dol00] Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
- [GM02] Godard, E., Métivier, Y.: A characterization of families of graphs in which election is possible. In: Nielsen, M., Engberg, U. (eds.) *FoSSaCS 2002*. LNCS, vol. 2303, pp. 159–171. Springer, Heidelberg (2002). doi:[10.1007/3-540-45931-6_12](https://doi.org/10.1007/3-540-45931-6_12)
- [GMT10] Godard, E., Métivier, Y., Tel, G.: Termination detection of local computations. Technical Report. [arXiv:1001.2785v2](https://arxiv.org/abs/1001.2785v2), January 2010
- [God02] Godard, E.: A self-stabilizing enumeration algorithm. *Inf. Process. Lett.* **82**(6), 299–305 (2002)
- [Maz88] Mazurkiewicz, A.: Solvability of the asynchronous ranking problem. *Inf. Process. Lett.* **28**, 221–224 (1988)
- [MMW97] Métivier, Y., Muscholl, A., Wacrenier, P.-A.: About the local detection of termination of local computations in graphs. In: Krizanc, D., Widmayer, P. (eds.), *SIROCCO 97–4th International Colloquium on Structural Information & Communication Complexity*, *Proceedings in Informatics*, pp. 188–200. Carleton Scientific (1997)
- [SSP85] Szymanski, B., Shy, Y., Prywes, N.: Terminating iterative solutions of simultaneous equations in distributed message passing systems. In: *Proceedings of the 4th Symposium of Distributed Computing*, pp. 287–292 (1985)
- [Tel00] Tel, G.: Introduction to distributed algorithms. Cambridge University Press, New York (2000)

8. Conclusion

Chapter Five Deployment

The aim of this chapter is to present various solutions to the deployment problem. The *deployment objective* is to optimally place a group of robots in an environment of interest. The approach taken here consists of identifying aggregate functions that measure the quality of deployment of a given network configuration and designing control and communication laws that optimize these measures.

The variety of algorithms presented in the chapter stems from two causes. First, different solutions arise from the interplay between the spatially distributed character of the coordination algorithms and the limited sensing and communication capabilities of the robotic network. As an example, different solutions are feasible when agents have range-limited communication capabilities or when agents have omnidirectional line-of-sight visibility sensors. Second, there is no universal notion of deployment. Different scenarios give rise to different ways of measuring what constitutes a good deployment. As an example, a robotic network might follow a different strategy depending on whether or not it has information about areas of importance in the environment: in the first case, by incorporating the knowledge on the environment; or in the second, by assuming a worst-case scenario, where important things can be happening precisely at the furthest-away location from the network configuration.

Our exposition here follows Cortés et al. (2004, 2005), and Cortés and Bullo (2005). Our approach makes extensive use of the multicenter functions from geometric optimization introduced in Chapter 2. It is not difficult to synthesize continuous-time gradient ascent algorithms using the smoothness results presented in Section 2.3,

and characterize their asymptotic convergence properties (as we ask the reader to do in Exercises [E2.14](#) and [E2.15](#)). However, following the robotic network model of Chapter 3, we are interested in discrete-time algorithms. In general, gradient ascent algorithms implemented in discrete time require the selection of appropriate step sizes that guarantee the monotonic evolution of the objective function. This is usually accomplished via line search procedures, (see e.g., [Bertsekas and Tsitsiklis](#),

1997). In this chapter, we show that the special geometric properties of the multicenter functions and their gradients allow us to identify natural target locations for the robotic agents without the need to perform any line search.

The chapter is organized as follows. In the first section, we formally define the notions of deployment via task maps and multicenter functions. In the next section, we present motion coordination algorithms to achieve each deployment task. Specifically, we introduce control and communication laws based on various notions of geometric centers. We present convergence and complexity results for the proposed algorithms, along with simulations illustrating our analysis. The third section presents various simulations of the proposed motion coordination algorithms. We end the chapter with three sections on, respectively, bibliographic notes, proofs of the results presented in the chapter, and exercises. Throughout the exposition, we make extensive use of proximity graphs, multicenter functions, and geometric optimization. The convergence and complexity analyses are based on the LaSalle Invariance Principle and on linear dynamical systems defined by Toeplitz matrices.

5.1 PROBLEM STATEMENT

Here, we introduce various notions of deployment. We assume that $S = (1, \dots, n, E_{\text{cmm}})$ is a uniform robotic network, where the robots' physical state space is a (simple convex) polytope $Q \subset \mathbb{R}^d$ that describes an environment of interest. We define our notions of deployment relying upon the geometric optimization problems discussed in Section 2.3. Loosely speaking, we aim to deploy the robots in

such a way as to optimize one of the multicenter functions, such as the expected-value multicenter function H_{exp} , the disk-covering multicenter function H_{dc} , or the sphere-packing multicenter function H_{sp} . Indeed, these functions can be interpreted as quality-of-service measures for different scenarios. In order to formally define the task maps encoding the deployment objective, we take the following approach: since the optimizers of these measures are critical points, and these critical points are network configurations that make the gradients vanish, we define the task map to take the true value at these configurations.

5.1.1 The distortion, area, and mixed distortion-area deployment tasks

In this section, we define various notions of deployment originating from the expected-value multicenter function H_{exp} . Recall the concepts of density and performance introduced in Section 2.3. Let $\varphi : \mathbb{R}^d \rightarrow \mathbb{R}_{>0}$ be a density function on $\mathbb{R}^d$ with support Q . One can interpret φ as a function measuring the probability that some event takes place over the environment. Let

$f: \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$ be a performance, that is, a non-increasing and piecewise differentiable function possibly with finite jump discontinuities. Performance functions describe the utility of placing a robot at a certain distance from a location in the environment. Here, we will restrict our attention to the cases $f(x) = -x^2$ (distortion problem), $f(x) = 1_{[0,a]}(x)$, $a \in \mathbb{R}_{>0}$ (area problem), and $f(x) = x^2 1_{[0,a]}(x) + a^2 1_{]a,+\infty[}(x)$, with $a \in \mathbb{R}_{>0}$ (mixed distortion-area problem).

For $\varepsilon \in \mathbb{R}_{>0}$, we define the ε -distortion deployment task $T_{\varepsilon\text{-distor-dply}}$:

$Q^n \rightarrow \{\text{true}, \text{false}\}$ by

$$T_{\varepsilon\text{-distor-dply}}(P) = \begin{cases} \text{true}, & \text{if } \|p^{[i]} - \text{CM}_{\phi}(V^{[i]}(P))\|_2 \leq \varepsilon, i \in \{1, \dots, n\}, \\ \text{false}, & \text{otherwise,} \end{cases}$$

where $V^{[i]}(P)$ denotes the Voronoi cell of robot i , and $\text{CM}_{\phi}(V^{[i]}(P))$ denotes its centroid computed according to ϕ (see Section 2.1). In other words, $T_{\varepsilon\text{-distor-dply}}$ is true for those network configurations where each robot is sufficiently close to the centroid of its Voronoi cell. According to Theorem 2.16, centroidal Voronoi configurations correspond to the critical points of the multicenter function H_{dist} .

For $r, \varepsilon \in \mathbb{R}_{>0}$, we define the ε - r -area deployment task $T_{\varepsilon\text{-}r\text{-area-dply}}: Q^n \rightarrow \{\text{true}, \text{false}\}$ as follows: we define $T_{\varepsilon\text{-}r\text{-area-dply}}(P) = \text{true}$ whenever

$$\int_{V^{[i]}(P) \cap \partial B(p^{[i]}, \bar{r})} n_{\text{out}}(q) \phi(q) dq \leq \varepsilon, \quad i \in \{1, \dots, n\},$$

$$\text{and we define } T_{\varepsilon\text{-}r\text{-area-dply}}(P) = \text{false} \text{ otherwise. Here, the}$$

denotes the outward normal vector to $B(p^{[i]}, \bar{r})$. In other words, $T_{\varepsilon\text{-}r\text{-area-dply}}$ is true for those network configurations where each agent is sufficiently close to a local maximum for the area of its $\bar{r}$ -limited Voronoi cell $V^{[i]}(P) \cap B(p^{[i]}, \bar{r})$.

$V^{[i]}(P) \cap B(p^{[i]}, \frac{r}{2})$ at fixed $V^{[i]}(P)$. According to Theorem 2.16, the $\frac{r}{2}$ -

limited area-centered Voronoi configurations correspond to the critical points of the multicenter function $H_{\text{area}, r}$.

Finally, for $r, \varepsilon \in \mathbb{R}_{>0}$, we define the ε - r -distortion-area deployment task

$T_{\varepsilon-r\text{-distor-area-dply}} : Q^n \rightarrow \{\text{true}, \text{false}\}$ by

$T_{\varepsilon-r\text{-distor-area-dply}}(P)$

$$= \begin{cases} \text{true}, & \text{if } \max_{i \in \{1, \dots, n\}} \left(\frac{\text{area}(V^{[i]}(P))}{\text{area}(V)} - \frac{r^2}{4} \right) \leq \varepsilon, \\ \text{false}, & \text{otherwise.} \end{cases}$$

In other words, $T_{\varepsilon-r\text{-distor-area-dply}}$ is true for those network configurations where each robot is sufficiently close to the centroid of its $\frac{r}{2}$ -limited Voronoi

cell. According to Theorem 2.16, $\frac{r}{2}$ -limited centroidal Voronoi configurations

are the critical points of the multicenter function $H_{\text{dist-area}, r}$.

5.1.2 The disk-covering and sphere-packing deployment tasks

Here, we provide two additional notions of deployment based on the multi-center functions H_{dc} and H_{sp} , respectively.

For $\varepsilon \in \mathbb{R}_{>0}$, the ε -disk-covering deployment task $T_{\varepsilon\text{-dc-dply}} : Q^n \rightarrow \{\text{true}, \text{false}\}$ is defined as

$$T_{\varepsilon\text{-dc-dply}}(P) = \begin{cases} \text{true,} & \text{if } \|p^{[i]} - \text{CC}(V^{[i]}(P))\|_2 \leq \varepsilon, \ i \in \{1, \dots, n\}, \\ \text{false,} & \text{otherwise.} \end{cases}$$

where $\text{CC}(V^{[i]}(P))$ denotes the circumcenter of the Voronoi cell of robot i . In other words, $\varepsilon\text{-dc-dply}$ is true for those network configurations where each robot is sufficiently close to the circumcenter of its Voronoi cell. Accord-

ing to Section 2.3.2, circumcenter Voronoi configurations are, under certain technical conditions, critical points of the multicenter function H_{dc} .

For $\varepsilon \in \mathbb{R}_{>0}$, the ε -sphere-packing deployment task $T_{\varepsilon\text{-sp-dply}} : Q^n \rightarrow \{\text{true}, \text{false}\}$ is defined as

$$T_{\varepsilon\text{-sp-dply}}(P) = \begin{cases} \text{true,} & \text{if } \text{dist}_2(p^{[i]}, \text{IC}(V^{[i]}(P))) \leq \varepsilon, \ i \in \{1, \dots, n\}, \\ \text{false,} & \text{otherwise.} \end{cases}$$

where $\text{IC}(V^{[i]}(P))$ denotes the incenter set of the Voronoi cell of robot i . In other words, $\varepsilon\text{-sp-dply}$ is true for those network configurations where each robot is sufficiently close to the incenter set of its Voronoi cell. According to Section 2.3.3, incenter Voronoi configurations are, under certain technical conditions, critical points of the multicenter function H_{sp} .

5.2 DEPLOYMENT ALGORITHMS

In this section, we present algorithms that can be used by a robotic network to achieve the various notions of deployment introduced in the previous section. Throughout the discussion, we use the uniform networks $\mathcal{S}_D$ and $\mathcal{S}_{LD}$ of locally connected first-order agents with the Delaunay and r -limited Delaunay communication, respectively, introduced in Example 3.4, and the uniform network $\mathcal{S}_{\text{vehicles}}$ of planar vehicle robots with Delaunay communication introduced in Example 3.5. The networks $\mathcal{S}_D$ and $\mathcal{S}_{LD}$ evolve in a polytope $Q \subseteq \mathbb{R}^d$, while the network $\mathcal{S}_{\text{vehicles}}$ evolves in a convex polygon $Q \subseteq \mathbb{R}^2$. For all the laws presented in this chapter, we assume that no two agents are initially at the same position, i.e., we assume that the initial

network configuration always belongs to Q^n_{coinc} , where n denotes the number of robots.

All the laws presented in this chapter share a similar structure, which we loosely describe as follows:

[Informal description] In each communication round, each agent performs the following tasks: (i) it transmits its position and receives its neighbors' positions; (ii) it computes a notion of the geometric center of its own cell, determined according to some notion of partition of the environment. Between communication rounds, each robot moves toward this center.

The notions of geometric center and of partition of the environment are different for each algorithm, and specifically tailored to the deployment task at hand. Let us examine them for each case.

5.2.1 Geometric-center laws

We present control and communication laws defined on the network D . All the laws share in common the use of the notion of Voronoi partition of the environment Q . We first introduce the $V_{rn}\text{-cntrd}$ law, which makes use of the notion of the centroid of a Voronoi cell. We then propose two sets of variations to this law. On the one hand, we present the $V_{rn}\text{-cntrd-dynmcs}$ law, which implements the same centroid strategy on a network of planar vehicles. On the other hand, we introduce the $V_{rn}\text{-circmctr}$ and $V_{rn}\text{-ncntr}$ laws, which instead make use of the notions of the circumcenter and incenter of a Voronoi cell, respectively.

5.2.1.1 Voronoi-centroid control and communication law

Here, we define the Vrn-cntrd control and communication law for the network $\mathcal{D}$, which we denote by Vrn-cntrd . This law was introduced by Cortés et al. (2004). We formulate the algorithm using the description model of Chapter 3. The law¹ is uniform, static, and data-sampled, with

¹From Definition 3.9 and Remark 3.11 recall that a control and coordination law (1) is uniform if processor state set, message-generation, state-transition and control functions are the same for each agent; (2) is static if the processor state set is a singleton, i.e., the law requires no memory;

(3) is data-sampled if the control functions are independent of the current position of the robot and depend only upon the robots position at the last sample time.

standard message-generation function:

Robotic Network: $\mathcal{S}_D$ with discrete-time motion model (4.1.1)
in Q , with absolute sensing of own position

Distributed Algorithm: Vrn-cntrd

Alphabet: $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$

function $\text{msg}(p, i)$

1: return p

function $\text{ctl}(p, y)$

1: $V := Q \cap \bigcap_{p_{\text{rcvd}} \in y} H_{p, p_{\text{rcvd}}}$ for all non-null $p_{\text{rcvd}} \in y$

2: return $\text{CM}_\phi(V) - p$

Recall that $H_{p,x}$ is the half-space of points q in $\mathbb{R}^d$ with the "property that $\|q - p\|_2 \leq \|q - x\|_2$. Since the centroid of a Voronoi cell belongs to the interior of the cell itself, if the robots are at distinct locations at any one time, then they are at distinct locations after one step. Therefore, the set Q^{coinc} is positively invariant with respect to the control and communication law $C^{\text{Vrn-cntrd}}$. Moreover, note that the direction of motion specified by the control function ctl coincides with the gradient of the distortion multicenter function H_{dist} . Hence, this law prescribes a gradient ascent strategy for each robot that, as we will show later, monotonically optimizes H_{dist} .

5.2.1.2 Voronoi-centroid law on planar vehicles

Next, we provide an interesting variation of the Vrn-cntrd law defined on the network $\mathcal{S}_{\text{vehicles}}$. Accordingly, we adopt the continuous-time motion model for the unicycle vehicle:

$$\dot{p}^{[i]}(t) = v^{[i]}(t) (\cos(\theta^{[i]}(t)), \sin(\theta^{[i]}(t))),$$

$$\ddot{\theta}^{[i]}(t) = \omega^{[i]}(t), \quad i \in \{1, \dots, n\}, \quad (5.2.1)$$

where we assume that forward and angular velocities are upper bounded. We refer to this control and communication law as the Vrn-cntrd-dynmcs law, and we denote it by Vrn-cntrd-dynmcs . The law was introduced by Cortés et al. (2004) and is uniform and static, but not data-sampled:

Robotic Network: vehicles with motion model (5.2.1) in Q ,
with absolute sensing of own position

Distributed Algorithm: Vrn-cntrd-dynmcs

Alphabet: $\mathbf{A} = \mathbb{R}^2 \cup \{\text{null}\}$

function $\text{msg}((p, \theta), i)$

1: return p

function $\text{ctl}((p, \theta), (p_{\text{smpld}}, \theta_{\text{smpld}}), y)$

1: $V := Q \cap \bigcap_{p_{\text{smpld}}, p_{\text{rcvd}} \in y} \{H_{p_{\text{smpld}}, p_{\text{rcvd}}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}$

2: $v := k_{\text{prop}} (\cos \theta, \sin \theta) \cdot (p - \text{CM}_{\phi}(V))$
 $\quad (\cos \theta, \sin \theta) \cdot (p - \text{CM}_{\phi}(V))$

3: $\omega := 2k_{\text{prop}} \arctan(\cos \theta, \sin \theta) \cdot (p - \text{CM}_{\phi}(V))$

4: return (v, ω)

This algorithm is illustrated in Figure 5.1.

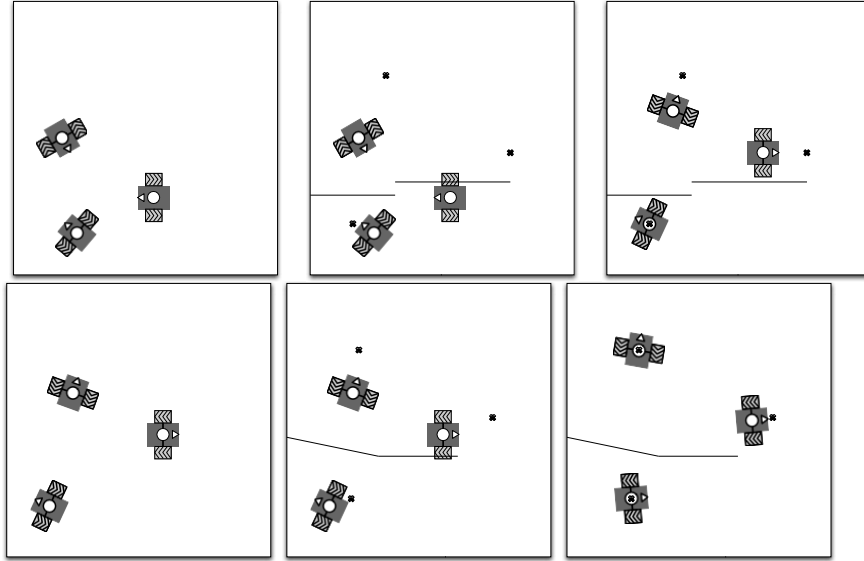


Figure 5.1 An illustration of the execution of Vrn-cntrd-dynmcs. Each row of plots represents an iteration of the law. In each round, each agent first computes its Voronoi cell, then determines the centroid, and then moves towards it.

In the above description, we require the feedback gain k_{prop} to belong to the interval $]0, \frac{1}{\max\{\pi, \text{diam}(Q)\}}]$. This guarantees that the controls v, ω in

the definition of ctl belong to the closed interval $[-1, 1]$, and are

therefore,
implementable in the unicycle and the differential drive robot models.

The definition of the control function ctl is based on the stabilizing feed-back law of [Astolfi \(1999\)](#). When following this control law, the robot position p is guaranteed to monotonically approach the target position $\text{CM}_\phi(V)$.

Unfortunately, it is only a conjecture that this controller (or an appropriately modified controller) does not lead two agents to the same positions (indeed, it is possible that an agent move outside its Voronoi cell). Under this conjecture, the Vrn-cntrd-dynmcs law enjoys the same convergence guarantees as the Vrn-cntrd law, that are described in Theorem 5.5.

Remark 5.1 (Vehicles with general dynamics). The general idea of moving towards the centroid of a robot's Voronoi region can be implemented over a network of vehicles with arbitrary dynamics, as long as these vehicles are capable of strictly decreasing the distance to any specified position in Q in the time intervals between communication rounds while remaining inside their Voronoi cells.

•

5.2.1.3 Voronoi-circumcenter control and communication law

Here, we define the Vrn-crcmctr control and communication law for the network $\mathcal{N}_D$, which we denote by Vrn-crcmctr . This law was introduced by Cortés and Bullo (2005). The law is uniform, static, and data-sampled, with standard message-generation function:

Robotic Network: $\mathcal{N}_D$ with discrete-time motion model (4.1.1) in Q , with absolute sensing of own position

Distributed Algorithm: Vrn-crcmctr

Alphabet: $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$

function $\text{msg}(p, i)$

1: return p

function $\text{ctl}(p, y)$

1: $V := Q \cap \bigcap_{p_{\text{rcvd}} \in y} H_{p, p_{\text{rcvd}}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}$

2: return $CC(V) - p$

Note that the circumcenter of a Voronoi cell belongs to the cell itself and therefore, robots evolving under the control and communication law

$CC_{\text{circumcenter}}$ never leave the set Q . However, in general the set $Q^n \setminus \text{coinc}$ is not positively invariant, see Exercise E5.1. From a geometric perspective, this law makes sense as a strategy to optimize the disk-covering multicenter function d_c . From Section 2.1.3, for fixed V , the circumcenter location minimizes the cost given by the maximum distance to all points in V . From Section 2.3.2, d_c can be expressed (2.3.12) as the maximum over the network of each robot's individual cost.

5.2.1.4 Voronoi-incenter control and communication law

Here, we define the $V_{rn-ncntr}$ control and communication law for the network $\mathcal{N}_D$, which we denote by $V_{rn-ncntr}$. This law was introduced by Cortés and Bullo (2005). The law is uniform, static, and data-sampled, with standard message-generation function:

Robotic Network: $\mathcal{N}_D$ with discrete-time motion model (4.1.1) in Q , with absolute sensing of own position

Distributed Algorithm: $V_{rn-ncntr}$

Alphabet: $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$

function $\text{msg}(p, i)$

1: return p

function $\text{ctl}(p, y)$

1: $V := Q \cap \bigcap_{p_{\text{rcvd}} \in y} \{H_{p, p_{\text{rcvd}}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}$

2: return $x \in \text{IC}(V) - p$

Since the incenter set of a Voronoi cell belongs to the interior of the cell itself, if the robots are at distinct locations at any one time, then they are at distinct locations after one step. That is, the set $Q^n \setminus S_{\text{coinc}}$ is

positively invariant with respect to the control and communication law $V_{rn-ncntr}$.

From a geometric perspective, this law makes sense as a strategy for op-

timizing the sphere-packing multicenter function sp .

From Section 2.1.3, for fixed V , the incenter locations maximize the cost given by the minimum distance to the boundary of V . From Section 2.3.3, sp can be expressed (2.3.15) as the minimum over the network of each robot's individual cost.

Remark 5.2 (“Move-toward-furthest-vertex” and “away-from-closest-neighbor” coordination algorithms). Consider the coordination algorithm

where, at each time step, each robot moves towards the furthest-away vertex of its own Voronoi cell. Alternatively, consider the coordination algorithm where, at each time step, each robot moves away from its closest neighbor. Both coordination algorithms define maps which depend discontinuously on the robots' positions. Cortés and Bullo (2005) study the asymptotic behavior of these laws, and show that the “move-toward-furthest-vertex” algorithm monotonically optimizes the multicenter function H_{dc} , while the “away-from-closest-neighbor” algorithm monotonically optimizes the multicenter function H_{sp} .

•

5.2.2 Geometric-center laws with range-limited interactions

In the following, we present two control and communication laws on the network $\mathcal{S}_{LD}$. Both laws prescribe a geometric centering strategy for each robot and accomplish specific forms of expected-value optimization. The Lmtd-Vrn-nrml law optimizes the area multicenter function $H_{\text{area}, r}^2$, while

the Lmtd-Vrn-cntrd law optimizes the mixed distortion-area multicenter function $H_{\text{dist-area}, r}$.

5.2.2.1 Limited-Voronoi-normal control and communication law

Here, we define the Lmtd-Vrn-nrml control and communication law for the network $\mathcal{S}_{LD}$. This law was introduced by Cortés et al. (2005). The Lmtd-Vrn-nrml law, which we denote by $\mathcal{C}_{\text{Lmtd-Vrn-nrml}, r}$, uses the notion of r -limited Voronoi partition inside Q . The law is uniform, static, and data-sampled, with standard message-generation function:

Robotic Network: $\mathcal{S}_{LD}$ with discrete-time motion model (4.1.1)
 $\mathcal{S}$
 with absolute sensing of own position, and with communication range r , in Q

Distributed Algorithm: Lmtd-Vrn-nrml

Alphabet: $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$

function $\text{msg}(p, i)$

1: return p

function $\text{ctl}(p, y)$

1: $V := Q \cap \bigcap_{p_{\text{rcvd}} \in y} \{H_{p, p_{\text{rcvd}}}^{-T} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}^\Sigma$

2: $v := \frac{\int_{V \cap \partial B(p, r)} n_{\text{out}}(q) \varphi(q) dq}{\bar{v}}$

3: $\lambda_* := \max \{ \lambda \in [0, \delta] \mid \int_{V \cap B(p + \delta v, \frac{r}{2})} \varphi(q) dq \text{ is strictly increasing on } [0, \lambda] \}$

4: return $\lambda_* v$

In the above algorithm, n_{out} denotes the outward normal vector to $B(p, r)$. Note that the direction of motion² v specified by the control function ctl co- incides with the gradient of the multicenter function $H_{\text{area}, r}$. The parameter λ_* corresponds to performing a line search² procedure along the direction of the vector v .

The control function has the property that the point $p + \text{ctl}(p, y)$ is guaranteed to be in the interior of V . This can be justified by noting that for fixed V , the gradient of the function $p \mapsto \int_{V \cap B(p, r)} \varphi(q) dq$ at points in the

boundary of V is non-vanishing and points toward the interior of V (cf.

Exercise E2.5). As a consequence, the line search procedure terminates before reaching the boundary of V . This discussion guarantees that the set $Q^n \setminus S_{\text{coinc}}$ is positively invariant with respect to the control and communication law $\text{CC}_{\text{Lmtd-Vrn-nrml}}$.

5.2.2.2 Limited-Voronoi-centroid control and communication law

Here, we define the Lmtd-Vrn-cntrd control and communication law for the network S_{LD} . This law was introduced by Cortés et al. (2005). The Lmtd-Vrn-cntrd law, which we denote by $\text{CC}_{\text{Lmtd-Vrn-cntrd}}$, uses the notion of r -limited Voronoi partition inside Q and of centroid of the individual

r -limited Voronoi cells. The law is uniform, static, and data-sampled, with standard message-generation function:

Robotic Network: S_{LD} with discrete-time motion model (4.1.1)
 S
 with absolute sensing of own position, and with communication range r , in Q

Distributed Algorithm: Lmtd-Vrn-cntrd

Alphabet: $\mathbf{A} = \mathbb{R}^d \cup \{\text{null}\}$

function $\text{msg}(p, i)$

1: return p

function $\text{ctl}(p, y)$

1: $V := Q \cap \overline{B}(p, \frac{r}{2}) \cap \{H_{p, p_{\text{rcvd}}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}$

2: return $\text{CM}_\phi(V) - p$

The centroid of a $\frac{r}{2}$ -limited Voronoi cell belongs to the interior of the cell

itself, and this fact guarantees that the set $Q \setminus S_{\text{coinc}}^n$ is positively invariant with respect to the control and communication law $\text{C}_{\text{Lmtd-Vrn-ctrld}}$. More-

over, note that the direction of motion specified by the control function ctl coincides with the gradient₂ of the multicenter function $H_{\text{dist-area}, r}$.

Remark 5.3 (Relative sensing version). It is possible to implement the limited-Voronoi-normal and limited-Voronoi-centroid laws as static relative-sensing control laws on the relative-sensing network S^{rs} . This is a consequence of the fact that the r -limited Delaunay graph is spatially distributed

over the r -disk graph (cf., Theorem 2.7(iii)). Let us present one of these examples for completeness:

Relative Sensing Network: $\mathbf{S}^{\text{rs}}$ with motion model (4.1.2) in Q , no communication, relative sensing for robot i given by:

robot measurements y contains $p^{[j]} \in B(0_d, r)$ for all $j \neq i$
 environment measurement is $y_{\text{env}} = (Q_\varepsilon)_i \cap B(0_d, r)$

Distributed Algorithm: relative-sensing Lmtd-Vrn-cntrd

function $\text{ctl}(y, y_{\text{env}})$

1: $V := y_{\text{env}} \cap B(0_d, r) \cap \bigcup_{p \in y} \{H_{0_d, p_{\text{snsd}}} \mid \text{for all non-null } p_{\text{snsd}} \in y\}$

2: return $\text{CM}_\phi(V)$

Note that only the positions of neighboring robots in the r -limited Delaunay graph have an effect on the computation of the set V .

•
 Remark 5.4 (Range-limited version of Vrn-cntrd). The Lmtd-Vrn-nrml and Lmtd-Vrn-cntrd laws can be combined into a single control and communication law to synthesize an algorithm that monotonically optimizes the function ϕ , with $b = \text{diam}(Q)^2$. This law, which we

term Rng-Vrn-cntrd, is uniform, static, and data-sampled, with standard message-generation function:

Robotic Network: $\mathbf{S}_{\text{LD}}$ with discrete-time motion model (4.1.1) in Q , with absolute sensing of own position, and with communication range r

Distributed Algorithm: Rng-Vrn-cntrd

Alphabet: $\mathbf{A} = \mathbf{R}^d \cup \{\text{null}\}$

```

function msg( $p, i$ )

1: return  $p$ 

function ctl( $p, y$ )

1:  $V := Q \cap \bigcap_{p \in y} \{H_{p, p_{\text{rcvd}}} \mid \text{for all non-null } p_{\text{rcvd}} \in y\}$ 
2:  $v_1 := 2 \frac{\text{diam}(Q)^2 - r^2}{4} \frac{\int_{V \cap \partial B(p, r)} n_{\text{out}}(q) \phi(q) dq}{\int_{V \cap \partial B(p, r)} \phi(q) dq}$ 
3:  $v_2 := \frac{\text{diam}(Q)^2 - r^2}{4}$ 
4:  $\lambda_* := \max \{ \lambda \mid \delta \mapsto H_V(p + \delta(v_1 + v_2), B(p + \delta(v_1 + v_2), \frac{r}{2})) \text{ is strictly increasing on } (0, \lambda) \}$ 
5: return  $\lambda_*(v_1 + v_2)$ 

```

In the above algorithm, n_{out} denotes the outward normal vector to $B(p, \frac{r}{2})$ and, for a point $p \in V$ and a closed ball $\bar{B}$ centered at a point in V with radius $\frac{r}{2}$, is defined as

$$H_V(p, \bar{B}) = - \int_{V \cap \bar{B}} \|q - p\|^2 \phi(q) dq - \text{diam}(Q)^2 \int_{V \cap (Q \setminus \bar{B})} \phi(q) dq.$$

The Rng-Vrn-cntrd law is relevant because of the following discussion. Recall from Proposition 2.17 that the general mixed distortion-area multicenter function can be used to provide constant-factor approximations of the distortion function dist . As we discussed in Section 2.3.1, robots with range-limited interactions cannot implement Vrn-cntrd because, for a given $r \in \mathbb{R}_{>0}$, dist is not in general spatially distributed over $\text{disk}(r)$ (cf., Remark 2.10). However, robotic agents with range-limited interactions can implement the computations involved in Lmtd-Vrn-nrml and Lmtd-Vrn-cntrd, and hence can optimize $H_{\text{dist-area}, r, b}$, with $b = -\text{diam}(Q)^2$. Assuming $r \leq 2 \text{diam}(Q)$, it is fair to say that the above algorithm can be understood as a range-limited version of the Vrn-cntrd law.

•

5.2.3 Correctness and complexity of geometric-center laws

In this section, we characterize the convergence and complexity properties of the geometric-center laws. The asynchronous execution of the Voronoi-centroid control and communication law can be studied as an asynchronous gradient dynamical system (see Cortés et al., 2004).

The following theorem summarizes the results known

in the literature about the asymptotic properties of these laws.

Theorem 5.5 (Correctness of the geometric-center algorithms). *For $d \in \mathbb{N}$, $r \in \mathbb{R}_{>0}$, and $\varepsilon \in \mathbb{R}_{>0}$, the following statements hold for any execution that starts from an configuration in $Q^n \setminus S_{\text{coinc}}$:*

- (i) *On the network S_D , the law $CC_{\text{Vim-entrd}}$ achieves the ε -distortion deployment task $T_{\varepsilon\text{-distor-dply}}$. Moreover, any execution of the law $CC_{\text{Vim-entrd}}$ monotonically optimizes the multicenter function H_{dist} .*
- (ii) *On the network S_D , any execution of the law $CC_{\text{Vim-crcmctr}}$ monotonically optimizes the multicenter function H_{dc} .*
- (iii) *On the network S_D , any execution of the law $CC_{\text{Vim-ncntr}}$ monotonically optimizes the multicenter function H_{sp} .*

- (i) On the network S_{LD} , the law $CC_{Lmtd-Vrn-nrml}$ achieves the ε - r -area deployment task $T_{\varepsilon-r-area-dply}$. Moreover, any execution of the law $CC_{Lmtd-Vrn-nrml}$ monotonically optimizes the multicenter function $H_{area, r}$.
- (ii) On the network S_{LD} , the law $CC_{Lmtd-Vrn-cntrd}$ achieves the ε - r -distortion-area deployment task $T_{\varepsilon-r-distor-area-dply}$. Moreover, any execution of $CC_{Lmtd-Vrn-cntrd}$ monotonically optimizes the multi-center function $H_{dist-area, r}$.

The proof of this theorem is given in Section 5.5.1. The results on $CC_{Vrn-cntrd}$ appeared originally in Cortés et al. (2004). (Note that an execution of $Vrn-cntrd$ can be viewed as an alternating sequence of configuration of points and partitions of the space, with the properties that (i) each configuration of points corresponds to the set of centroid locations of the immediately preceding partition in the sequence, and (ii) each partition corresponds to the Voronoi partition determined by the immediately preceding configuration of points in the sequence. The monotonic behavior of H_{dist} now follows from Propositions 2.13 and 2.14. Similar interpretations can be given to all other laws. In particular, the monotonic behavior of H_{dc} along executions of $CC_{Vrn-crcmctr}$ can be established via Proposition 2.19, and the monotonic behavior of H_{sp} along executions of $Vrn-ncntr$ can be established via Proposition 2.21. Continuous-time versions of these laws are studied by Cortés and Bullo (2005) via nonsmooth stability analysis, where the following convergence properties are established (recall the notion of active and passive nodes introduced in Sections 2.3.2 and 2.3.3): all active agents are guaranteed to asymptotically reach the circumcenter (resp., incenter) of their Voronoi region, whereas it is not known if the same conclusion holds for the passive agents. Depending on the polytope Q , there exist circumcenter and incenter Voronoi configurations where not all agents are active, and simulations show that in some cases the continuous-time

versions of $CC_{Vrn-crcmctr}$ and $CC_{Vrn-ncntr}$ converge to them. It is an open research question to show that $CC_{Vrn-crcmctr}$ and $CC_{Vrn-ncntr}$ achieve the ε -disk-covering deployment task $T_{\varepsilon-dc-dply}$ and the ε -sphere-packing deployment task $T_{\varepsilon-sp-dply}$, respectively. Finally, the results on $CC_{Lmtd-Vrn-nrml}$ and $CC_{Lmtd-Vrn-cntrd}$ appeared in [Cortés et al. \(2005\)](#).

Next, we analyze the time complexity of $CC_{Lmtd-Vrn-cntrd}$. We provide complete results only for the case $d = 1$ and uniform density. We assume that $\text{diam}(Q)$ is independent of n , r , and ε .

Theorem 5.6 (Time complexity of $Lmtd-Vrn-cntrd$ law). *Assume that the robots evolve in a closed interval $Q \subset \mathbb{R}$, that is, $d=1$, and assume that the density is uniform, that is, $\varphi \equiv 1$. For $r \in \mathbb{R}_{>0}$ and $\varepsilon \in \mathbb{R}_{>0}$, on the network S_{LD} , $TC(T_{\varepsilon-r-distor-area-dply}, CC_{Lmtd-Vrn-cntrd}) \in O(n^3 \log(n\varepsilon^{-1}))$.*

The proof of this result is given in Section 5.5.2 following the treatment in Martínez et al. (2007b).

Remark 5.7 (Congestion effects). Interestingly, Theorem 5.6 also holds if, motivated by wireless congestion considerations, we take the communication range r to be a monotone non-increasing function $r : \mathbb{N} \rightarrow]0, 2\pi[$ of the number of robotic agents n .

•

5.3 SIMULATION RESULTS

In this section, we illustrate the execution of the various control and communication laws introduced in this chapter.

Geometric-center algorithms for expected-value optimization

The Vrn-cntrd, Lmtd-Vrn-nrml, and Lmtd-Vrn-cntrd control and communication laws are implemented in Mathematica[®] as a library of routines and a main program running the simulation. The objective of a first routine is to compute the r -limited Voronoi partition and parameterize each cell $V_r, i = 1, \dots, n$ in polar coordinates. The objective of a second routine is to compute the surface integrals on these sets and the line integrals on their boundaries via the numerical integration routine NIntegrate. We pay careful attention to numerical accuracy issues in the computation of the Voronoi diagram and in the integration.

Measuring displacements in meters, we consider the polygon Q determined by the vertices

$$\{(0, 0), (2.125, 0), (2.9325, 1.5), (2.975, 1.6),$$

$(2.9325, 1.7), (2.295, 2.1), (0.85, 2.3), (0.17, 1.2)\}$.

The diameter of Q is $\text{diam}(Q)$ 3.378.

In all figures, the density function φ is the sum of four Gaussian functions of the form $\frac{1}{11} \exp(-6((x - x_{\text{center}})^2 + (y - y_{\text{center}})^2))$ and is represented by means of its contour plot.

Darker-colored areas correspond to higher values of the density function. The

four centers $(x_{\text{center}}, y_{\text{center}})$ of the Gaussian functions are the points $(2.15, 0.75), (1.0, 0.25), (0.725, 1.75)$ and $(0.25, 0.7)$, respectively. The area of the polygon is $A_\phi(Q) = 17.6352$.

We show evolutions of $(\mathcal{D}, \text{Vrn-cntrd})$ and $(\mathcal{D}, \text{Vrn-cntrd-dynmcs})$ in Figures 5.2 and 5.3, respectively. One can verify that the final network configurations is a centroidal Voronoi configuration. In other words, the task $T_{\varepsilon\text{-distor-dply}}$ is achieved, as guaranteed by Theorem 5.5(i) for the Vrn-

cntrd algorithm. For each evolution we depict the initial positions, the trajectories, and the final positions of all robots.

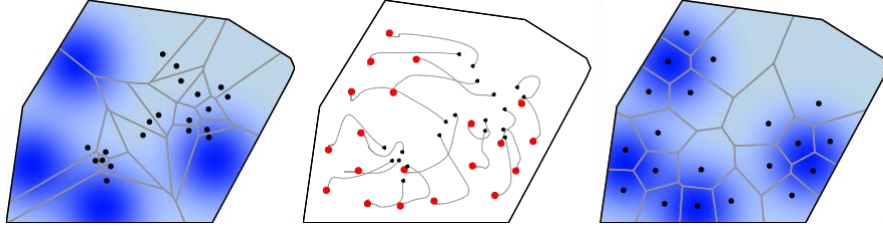


Figure 5.2 The evolution of $(S_D, \text{Vrn-cntrd})$ with $n = 20$ robots. The left-hand (resp., right-hand) figure illustrates the initial (resp., final) locations and Voronoi partition. The central figure illustrates the evolution of the robots. After 13 seconds, the value of H_{dist} has monotonically increased to approximately -0.515 .

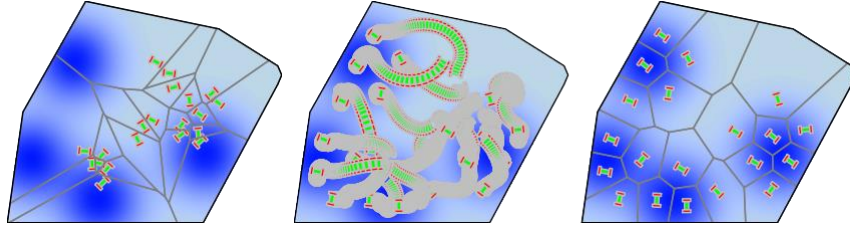


Figure 5.3 The evolution of $(S_D, \text{Vrn-cntrd-dynmcs})$ with $n = 20$ robots and with feedback gain $k_{\text{prop}} = 3.5$. The left-hand (resp., right-hand) figure illustrates the initial (resp., final) locations and Voronoi partition. The central figure illustrates the evolution of the robots. After 20 seconds, the value of H_{dist} has monotonically increased to approximately -0.555 .

We show an evolution of $(S_{LD}, \text{Lmtd-Vrn-nrml})$ in Figure 5.4. One can verify that the final network configuration is an r -limited area-centered Voronoi configuration. In other words, the task ε - r -area-dply is achieved, as guaranteed by Theorem 5.5(ii).

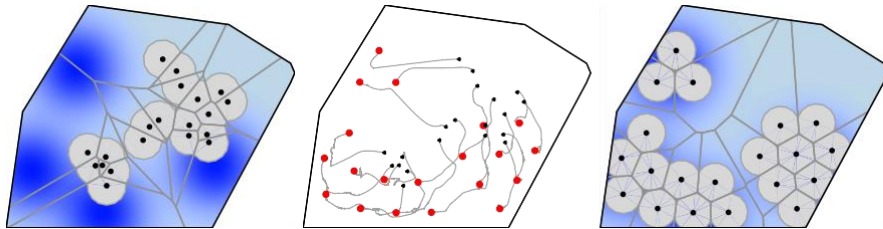


Figure 5.4 The evolution of $(S_{LD}, \text{Lmtd-Vrn-nrml})$ with $n = 20$ robots and $r = 0.4$. The left-hand (resp., right-hand) figure illustrates the initial (respectively, final) locations and Voronoi partition. The central figure illustrates the evolution of the robots. The $\frac{\varepsilon}{2}$ -limited Voronoi cell of each robot is plotted in light

gray. After 36 seconds, the value of $H_{\text{area}, \frac{\epsilon}{2}}$ has monotonically increased to

approximately 14.141.

We show an evolution of $(S_{LD}, \text{Lmtd-Vrn-cntrd})$ in Figure 5.5. One can verify that the final network configuration is a r -limited ^{2} centroidal Voronoi configuration. In other words, the task ε - r -distor-area-dply is achieved, as guaranteed by Theorem 5.5(iii).

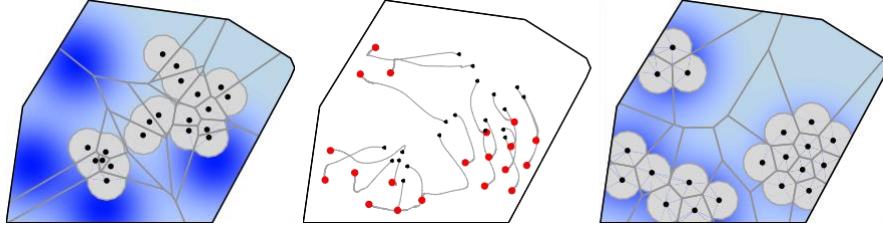


Figure 5.5 The evolution of $(S_{LD}, \text{Lmtd-Vrn-cntrd})$ with $n = 20$ robots and $r = 0.4$. The left-hand (resp., right-hand) figure illustrates the initial (resp., final) locations and Voronoi partition. The central figure illustrates the evolution of the robots. The r -limited Voronoi cell of each robot is plotted in light gray. After 90 seconds, the value of $H_{\text{dist-area}, r, \frac{r}{2}}$ reaches approximately -0.386 .

We show an evolution of $(S_{LD}, \text{Rng-Vrn-cntrd})$ in Figure 5.6. One can verify that the final network configuration corresponds to a critical point of the mixed distortion-area multicenter function $H_{\text{dist-area}, r, b}$, with $b = -\text{diam}(Q)^2$, (see Exercise E5.4).

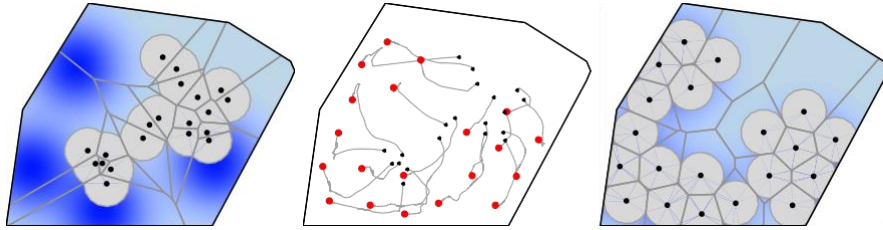


Figure 5.6 The evolution of $(S_{LD}, \text{Rng-Vrn-cntrd})$ with $n = 20$ robots and $r = 0.47$. The left-hand (resp., right-hand) figure illustrates the initial (respectively, final) locations and Voronoi partition. The central figure illustrates the evolution of the robots. The r -limited Voronoi cell of each robot is plotted in light gray. After 13 seconds, the value of $H_{\text{dist-area}, r, b, \frac{r}{2}}$ with $b = -\text{diam}(Q)^2$, is approximately -4.794 .

As discussed in Remark 5.4, Rng-Vrn-cntrd can be understood as a range-limited implementation of Vrn-cntrd in a network of robots with range-limited

interactions. Let us briefly compare the evolutions depicted in Figures 5.2 and 5.6. According to Proposition 2.17, we compute

$$\beta = \frac{\frac{r}{2}}{\text{diam } Q} \approx 0.06957.$$

From the constant-factor approximation (2.3.7), the absolute error is guaranteed to be less than or equal to $(\beta^2 - 1)H_{\text{dist-area}, r, b}(P_{\text{final}}) \approx 4.77$, where P_{final} denotes the final configuration in Figure 5.6. The percentage error in the value of the multicenter function dist between the final configuration of the evolution in Figure 5.2 and the final configuration of the evolution in Figure 5.6 is approximately equal to 3.277%. As expected, one can verify in simulations that the percentage error of the performance of the range-limited implementation improves with higher values of the ratio $\frac{r}{d_{\text{min}}}$.

Geometric-center algorithms for disk-covering and sphere-packing

The `Vrn-crcmctr` and `Vrn-ncntr` control and communication laws are implemented in Mathematica[®] as a single centralized program running the simulation. We compute the bounded Voronoi diagram of a collection of points using the package `ComputationalGeometry`. We compute the circumcenter of a polygon via the algorithm in [Skyum \(1991\)](#) and the incenter set via the `LinearProgramming` solver in Mathematica[®].

Measuring displacements in meters, we consider the polygon determined by the vertices

$$\{(0, 0), (2.5, 0), (3.45, 1.5), (3.5, 1.6), (3.45, 1.7), (2.7, 2.1), (1.0, 2.4), (0.2, 1.2)\}.$$

We show an evolution of $(D, \text{Vrn-crcmctr})$ in Figure 5.7. One can verify that in the final configuration all robots are at the circumcenter of their own Voronoi cell. In other words, the task $\varepsilon\text{-dc-dply}$ is achieved by this evolution. As stated in Section 5.2.3, it is an open research question to show

that this fact holds in general for Vrn-crcmctr . Cortés and Bullo (2005) prove a similar result for a continuous-time implementation of this law.

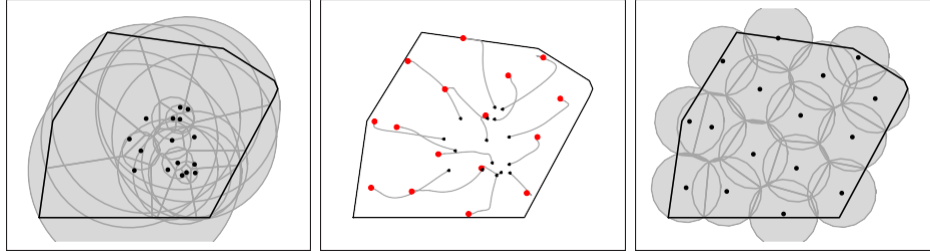


Figure 5.7 The evolution of $(S_D, \text{Vrn-crcmctr})$ with $n = 16$ robots. The left-hand (resp., right-hand) figure illustrates the initial (resp., final) locations and Voronoi partition. The central figure illustrates the evolution of the robots. After 20 seconds, the value of H_{dc} has monotonically decreased to approximately 0.43273 meters.

We show an evolution of $(S_D, \text{Vrn-nctr})$ in Figure 5.8. One can verify that in the final configuration all robots are at the incentre of their own Voronoi cell. In other words, the task $\varepsilon\text{-sp-dply}$ is achieved by this evolution. As stated in Section 5.2.3, it is an open research question to show that this fact holds in general for Vrn-nctr . Cortés and Bullo (2005) prove a similar result for a continuous-time implementation of this law.

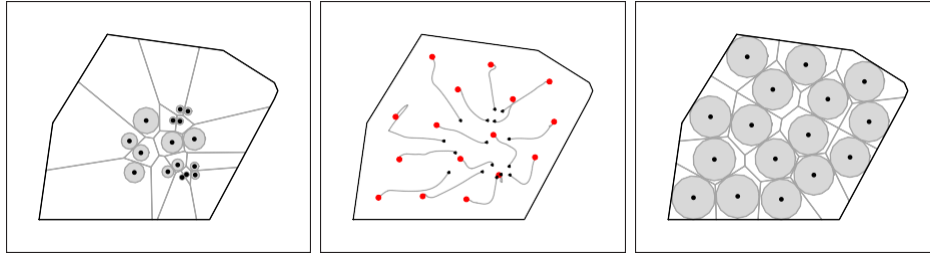


Figure 5.8 The evolution of $(S_D, \text{Vrn-nctr})$ with $n = 16$ robots. The left-hand (resp., right-hand) figure illustrates the initial (resp., final) locations and Voronoi partition. The central figure illustrates the evolution of the robots. After 20 seconds, the value of H_{sp} has monotonically increased to approximately 0.2498

meters.

5.4 NOTES

The deployment problem studied in this chapter is related to the literature on facility location (Drezner, 1995; Okabe et al., 2000; Du et al., 1999) and geometric optimization (Agarwal and Sharir, 1998; Boltyanski et al., 1999) (see also Section 2.4). These disciplines study spatial resource allocation problems and play an important role in quantization theory, mesh and grid optimization methods, clustering analysis, data compression, and statistical pattern recognition.

Dispersion laws have been traditionally studied in behavior control (see, e.g., (Arkin, 1998; Schultz and Parker, 2002; Balch and Parker, 2002)). Deployment algorithms that make use of potential field methods are

proposed by [Payton et al. \(2001\)](#) and [Howard et al. \(2002\)](#). Other works include [Bulusu et al., 2001](#)) on adaptive beacon placement for localization, [Poduri and Sukhatme \(2004\)](#) on network deployments that satisfy a pre-specified constraint in the number of neighbors of each robot, [Arsie et al. \(2009\)](#) on sensor-based deployment strategies that minimize the expected service time for newly appearing target points, and [Hussein and Stipanović \(2007\)](#) on dynamically surveying a known environment.

Deployment algorithms for coverage control are a subject of active re- search. Among the most recent works, [Martínez \(2009a\)](#) and [Schwager et al. \(2009\)](#) consider coverage problems where the density function is un-

known, [Lekien and Leonard \(2009\)](#) propose centralized laws for non-uniform coverage using cartograms, [de Silva and Ghrist \(2007\)](#) study static coverage problems with minimal assumptions on the capabilities of individual sensors using algebraic topology, [Kwok and Martínez \(2009\)](#) propose distributed deployment strategies for energy-constrained networks, [Laventall and Cortés \(2009\)](#) design distributed algorithms for networks of robots whose sensors have range-limited wedge-shaped footprints, [Gao et al. \(2008\)](#) consider discrete coverage problems, [Schwager et al. \(2008\)](#) consider joint exploration and deployment problems, and [Zhong and Cassandras \(2008\)](#), [Pimenta et al. \(2008\)](#), and [Caicedo-Núñez and Zefran \(2008\)](#) deal with centroidal Voronoi tessellations in nonconvex environments. [Graham and Cortés \(2009\)](#) study the optimality of circumcenter and incenter Voronoi configurations for the estimation of stochastic spatial fields. [Susca et al. \(2009\)](#) consider some planar interpolation problems.

Deployment problems play a relevant role in other coordination tasks, such as surveillance, search and rescue, and exploration and map building of unknown environments. [Choset \(2001\)](#) considers sweep coverage problems, where one or more robots equipped with limited footprint sensors have to visit all points in the environment. In [Simmons et al. \(2000\)](#), deployment locations for a network of heterogeneous robots are user-specified after an initial map of the unknown environment has been built. [Gupta et al. \(2006a\)](#) consider a combined sensor coverage and selection problem.

Deployment of robotic agents with visibility sensors has been studied under a variety of assumptions. When the environment is known *a priori*, the problem can be cast as the classical Art Gallery Problem ([Chvátal, 1975](#)) from computational geometry, where one is interested in achieving complete visibility with the minimum number of agents possible. The Art Gallery Problem is computationally hard ([Lee and Lin, 1986](#); [Eidenbenz et](#)

al., 2001) and the best-known approximation algorithms yield solutions within a log-arithmic factor of the optimum number of agents (Ghosh, 1987; Efrat and Har-Peled, 2006). Pinciu (2003) and Hernández-Peñalver (1994) study the problem of achieving full visibility while guaranteeing that the final network configuration will have a connected visibility graph. Recent works on multi-robot exploration of unknown environments include (Batalin and Sukhatme, 2004), Burgard et al. (2005), and Howard et al. (2006). Topological exploration of graph-like environments by single and multiple robots is studied in Rekleitis et al. (2001), Fraigniaud et al. (2004), and Dynia et al. (2006). A simple one-step strategy for visibility deployment, without the need for synchronization, achieving the worst-case optimal bounds in terms of the number of robots required, and under limited communication, is presented in Ganguli et al. (2007).

5.5 PROOFS

This section gathers the proofs of the main results presented in the chapter.

5.5.1 Proof of Theorem 5.5

Proof. Let $P_0 = (p^{[1]}(o), \dots, p^{[n]}(o))$ denote the initial condition. The proof strategy for all five facts is similar and is based on the application of the LaSalle Invariance Principle with Lyapunov function given by an appropriate multicenter function. In other words, we need to establish the monotonic behavior of the certain multicenter functions along the executions of the control and communication laws and we need to characterize certain invariant sets using geometric properties of the multicenter functions. Additionally, in order to apply the LaSalle Invariance Principle, we need to work with the set Q^n_{coinc} which is not closed and, therefore, we rely upon the extension of Theorem 1.19 given in Exercise E1.8(ii).

In what follows, we discuss in detail the proof of fact (i) regarding the control and communication law $\text{CC}_{\text{Vrn-ctrl}}$ for the network $\mathcal{D}$. We leave it to the reader to fill out some of the proof details for the other laws.

Fact (i). First, note that, starting from a configuration in $Q^n \setminus S_{\text{coinc}}$, one step of the law $\text{CC}_{\text{Vrn-ctrl}}$ leads the network to another configuration in $Q^n \setminus S_{\text{coinc}}$. Therefore, it is convenient to let $f_{\text{Vrn-ctrl}} : Q^n \setminus S_{\text{coinc}} \rightarrow Q^n \setminus S_{\text{coinc}}$ denote the map induced by the execution of one step of the control and communication law $\text{CC}_{\text{Vrn-ctrl}}$.

To apply Theorem 1.19, we work with the set $W = Q^n \setminus S_{\text{coinc}}$. Clearly, this set is positively invariant for $f_{\text{Vrn-ctrl}}$ and it is bounded. Therefore,

assumptions (i) and (iii) of Theorem 1.19 are satisfied, except for the closed-ness of W . Next, we show that executions of the law $\mathcal{CC}_{\text{Vrn-cntrd}}^H$ monotonically optimize the function dist . Using the extension of the multicenter function defined over the set of points and partitions of Q , we deduce from

$$\begin{aligned} \text{Proposition 2.13 that, for } P \in Q^n \setminus S_{\text{coinc}}, \\ H_{\text{dist}}(f_{\text{Vrn-cntrd}}(P)) &= H_{\text{dist}}(f_{\text{Vrn-cntrd}}(P), V(f_{\text{Vrn-cntrd}}(P))) \\ &\geq H_{\text{dist}}(f_{\text{Vrn-cntrd}}(P), V(P)). \end{aligned}$$

The application of Proposition 2.14 yields

$$H_{\text{dist}}(f_{\text{Vrn-cntrd}}(P), V(P)) \geq H_{\text{dist}}(P, V(P)),$$

and, therefore, $H_{\text{dist}}(f_{\text{Vrn-cntrd}}(P)) \geq H_{\text{dist}}(P)$. Additionally, recall from Proposition 2.14, that, for $P \in Q^n \setminus S_{\text{coinc}}$, the inequality is strict unless $f_{\text{Vrn-cntrd}}(P) = P$. This discussion establishes assumption (ii) of Theo-

rem 1.19. The continuity of the map $f_{V_{\text{m-centrd}}} : Q^n \setminus S_{\text{coinc}} \rightarrow Q^n$ is a consequence of the following two facts. First, one can verify that each Voronoi cell is a convex set whose boundary is a piecewise continuously differentiable function of the positions of the robots. Second, given a convex set whose boundary depends upon a parameter in a piecewise continuously differentiable fashion, Proposition 2.23 guarantees that the centroid of that set is a continuously differentiable function of the parameter. This discussion and the continuity of H_{dist} establishes assumption (iv) of Theorem 1.19.

In the following, we consider an evolution $\gamma : Z_{\geq 0} \rightarrow Q^n \setminus S_{\text{coinc}}$ of $f_{V_{\text{m-centrd}}}$ and we prove that no point in S_{coinc} may be an accumulation point of γ . By contradiction, we assume that $P = (p_1, \dots, p_n) \in S_{\text{coinc}}$ is an accumulation point for γ . Our first claim is that there exists a sequence of increasing times $\{\ell_k \mid k \in \mathbb{N}\}$ and unit-length vectors $u_{ij} \in \mathbb{R}^d$, for $i, j \in \{1, \dots, n\}$, such that $\gamma(\ell_k) \rightarrow P$ and simultaneously $\text{vers}(\gamma_i(\ell_k) - \gamma_j(\ell_k)) \rightarrow u_{ij}$ as $k \rightarrow \infty$. Here, the versor operator $\text{vers} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ is defined by $\text{vers}(0_d) = 0_d$ and $\text{vers}(v) = v / \|v\|_2$ for $v \neq 0_d$. This first claim is true because P is an accumulation point and because the sequences

$\ell \mapsto \text{vers}(\gamma_i(\ell) - \gamma_j(\ell))$ take value in a compact set. Our second claim is that, as $k \rightarrow \infty$, the sequence of partitions $V(\gamma(\ell_k))$ has a limiting partition, say

$\{V_1^\infty, \dots, V_n^\infty\}$. This second claim is true because, for each pair of robots i

and j converging to the same position $p_i = p_j$, the bisector of the segment connecting them admits a limit that is equal to the line through the point

$p_i = p_j$ and perpendicular to the unit-length vector u_{ij} . Therefore, each of the edges of each of the polygons $V(\gamma(\ell_k))$ has a limit for $k \rightarrow \infty$.

Finally, note that each polygon V_i^∞ has a positive measure.

We know that $\ell_{\text{dist}}(\gamma(\ell))$ is monotonically non-

□

increasing and lower- bounded. Therefore, we must have that

$$\lim_{\ell \rightarrow \infty} \left[H_{\text{dist}}(\gamma(\ell)) - H_{\text{dist}}(\gamma(\ell + 1)) \right] = 0. \quad (5.5.1)$$

Define the short-hand $W_{k,1} = V_1(\gamma(\ell_k))$,

$$\begin{aligned} & \text{and compute } \lim_{k \rightarrow \infty} \left[H_{\text{dist}}(\gamma(\ell_k)) - \right. \\ & \left. H_{\text{dist}}(\gamma(\ell_k + 1)) \right] \\ & \geq \lim_{k \rightarrow \infty} \left[\int_{W_{k,1}} \frac{\|q - \gamma_1(\ell_k)\|^2}{2} \varphi(q) dq - \int_{W_{k,1}} \frac{\|q - \text{CM}_\phi(W_{k,1})\|^2}{2} \varphi(q) dq \right] \end{aligned} \quad (5.5.2)$$

$$= \int_{V_1^\infty} \frac{\|q - p_1\|^2}{2} \varphi(q) dq - \int_{V_1^\infty} \frac{\|q - \text{CM}_\phi(V_1^\infty)\|^2}{2} \varphi(q) dq \quad (5.5.3)$$

$$= A_\phi(V_1^\infty) \|p_1 - \text{CM}_\phi(V_1^\infty)\|_2^2, \quad (5.5.4)$$

where inequality (5.5.2) follows from Proposition 2.14, equality (5.5.3) follows from the definition of the limiting partition $\{V_1^\infty, \dots, V_n^\infty\}$, and equa-

tion (5.5.4) follows from the Parallel Axis Theorem. Now, the quantity $A_\phi(V_1^\infty)$ is strictly positive, as mentioned above, and the quantity $CM_\phi(V_1^\infty)$ is strictly positive because $p_1 \in \partial V_1^\infty$ and $CM_\phi(V_1^\infty)$ belongs to the interior of V_1^∞ by Exercise (E2.2). The fact that the last limit is lower bounded by a positive is a contradiction with equation (5.5.1). Therefore, we now know that no point in S_{coinc} may be an accumulation point of γ .

Finally, we are now ready to apply the LaSalle Invariance Principle as stated in Exercise E1.8(ii) and deduce that the execution of $f_{\text{Vrn-cntrd}}$ starting from $P_0 \in Q^n \setminus S_{\text{coinc}}$ tends to the largest positively invariant set S contained in

$$\{P \in Q^n \mid H_{\text{dist}}(f_{\text{Vrn-cntrd}}(P)) = H_{\text{dist}}(P)\}.$$

The set S is precisely the set of centroidal Voronoi configurations. This result is a consequence of the fact that $H_{\text{dist}}(f_{\text{Vrn-cntrd}}(P)) = H_{\text{dist}}(P)$ implies that $f_{\text{Vrn-cntrd}}(P) = P$, that is, P is a centroidal Voronoi configuration.

Facts (ii) and (iii). The proofs of these facts run parallel to the proof of fact (i). Propositions 2.19 and 2.21 are key in establishing the monotonic evolution of H_{dc} and H_{sp} , respectively.

Fact (iv). Let $f_{\text{Lmtd-Vrn-nrml}} : Q^n \setminus S_{\text{coinc}} \rightarrow Q^n \setminus S_{\text{coinc}}$ denote the map induced by the execution of one step of the law $\text{CC}_{\text{Lmtd-Vrn-nrml}}$. Let us show that executions of $f_{\text{Lmtd-Vrn-nrml}}$ monotonically optimize the function $H_{\text{area}, \frac{r}{2}}$. Using the extension of the multicenter function defined over

the set of points and partitions of Q , we deduce from Proposition 2.13 that, for $P \in Q^n \setminus S_{\text{coinc}}$,

$$\begin{aligned} H_{\text{area}, \frac{r}{2}}(f_{\text{Lmtd-Vrn-nrml}}(P)) \\ = H_{\text{area}, \frac{r}{2}}(f_{\text{Lmtd-Vrn-nrml}}(P), V(f_{\text{Lmtd-Vrn-nrml}}(P))) \end{aligned}$$

$$\geq H_{\text{area}, \frac{\varepsilon}{2}}(f_{\text{Lmtd-Vrn-nrml}}(P), V(P)).$$

The line search procedure for each robot embedded in the definition of the control function of $\text{CC}_{\text{Lmtd-Vrn-nrml}}$ ensures that

$$H_{\text{area}, \frac{\varepsilon}{2}}(f_{\text{Lmtd-Vrn-nrml}}(P), V(P)) \geq H_{\text{area}, \frac{\varepsilon}{2}}(P, V(P)),$$

and hence, $H_{\text{area}, \varepsilon}(f_{\text{Lmtd-Vrn-nrml}}(P)) \geq H_{\text{area}, \varepsilon}(P)$. Note that the inequality is strict unless $f_{\text{Lmtd-Vrn-nrml}}(P) = P$. We leave it to the interested reader to prove, similarly to what we did for fact (i), that the map $f_{\text{Lmtd-Vrn-nrml}}$ is continuous and that no point in S_{coinc} may be an accumulation point of any trajectory of $f_{\text{Lmtd-Vrn-nrml}}$. Finally, the application of the LaSalle Invariance Principle as in the proof of fact (i) leads us to the result stated in fact (iv).

Fact (v). The proof of this fact runs parallel to the proofs of facts (i)

and (iv). Propositions 2.13 and 2.15 are key in establishing the monotonic evolution of the cost function $H_{\text{dist-area}}$.

□

5.5.2 Proof of Theorem 5.6

Proof. For $d = 1$, Q is a compact interval on $\mathbb{R}$ —say $Q = [q_-, q_+]$. We start with a brief discussion about connectivity. In the r -limited Delaunay graph, two agents that are at most at a distance r from each other are neighbors if and only if there are no other agents between them. Additionally, we claim that if agents i and j are neighbors, then $|\text{CM}_\phi(V^{[i]}) - \text{CM}_\phi(V^{[j]})|$

$\leq r$, where

$V^{[i]}$ denotes the set defined by the control function ctl when evaluated by agent i . To show this fact, let us assume without loss of generality that

$p^{[i]} \leq p^{[j]}$. Let us consider the case where the agents have neighbors on both sides (the other cases can be treated analogously). Let $p_-^{[i]}$ (resp., $p_+^{[j]}$)

denote the position of the neighbor of agent i to the left (resp., of agent j to the right). Now,

$$\text{CM}_\phi(V^{[i]}) = \frac{1}{4}(p_-^{[i]} + 2p^{[i]} + p^{[j]}),$$

$$\text{CM}_\phi(V^{[j]}) = \frac{1}{4}(p^{[i]} + 2p^{[j]} + p_+^{[j]}),$$

+

where we have used the fact that $\phi \equiv 1$. Therefore,

$$|\text{CM}_\phi(V^{[i]}) - \text{CM}_\phi(V^{[j]})| \leq \frac{1}{4} \cdot |p_-^{[i]} - p^{[i]}| + 2|p^{[i]} - p^{[j]}| + |p^{[j]} - p_+^{[j]}| \leq r.$$

This implies that agents i and j belong to the same connected component of the r -limited Delaunay graph at the next time step.

Next, let us consider the case when $\text{LD}(r)$ is connected at the initial network configuration $P_0 = (p^{[1]}(o), \dots, p^{[n]}(o))$. Without loss of generality, assume that the agents are ordered from left to right according to their unique identifier, that is, $p^{[1]}(o) \leq p^{[n]}(o)$. We distinguish three cases depending on the proximity of the leftmost and rightmost agents 1 and n , respectively, to the boundary of the environment: in case (a) both agents are within a distance $\frac{r}{2}$ of ∂Q ; in case (b), neither of the two is within a distance $\frac{r}{2}$

of ∂Q ; and in case (c) only one of the agents is within a distance $\frac{r}{2}$ of ∂Q . Here is an important observation: from one time instant to the next,

the network configuration can fall into any of the cases described above. However, because of the discussion on connectivity, transitions can only occur from case (b) to either case (a) or case (c); and from case (c) to case (a).

As we show below, for each of these cases, the network evolution under Vrn-cntrd can be described as a discrete-time linear dynamical system which respects agents' ordering.

Let us consider case (a). In this case, we have

$$p^{[1]}(\ell + 1) = \frac{1}{4}(p^{[1]}(\ell) + p^{[2]}(\ell)) + \frac{1}{2}q_{-},$$

$$p^{[2]}(\ell + 1) = \frac{1}{4}(p^{[1]}(\ell) + 2p^{[2]}(\ell) + p^{[3]}(\ell)),$$

.

$$p^{[n-1]}(\ell + 1) = \frac{1}{4}(p^{[n-2]}(\ell) + 2p^{[n-1]}(\ell) + p^{[n]}(\ell)),$$

$$p^{[n]}(\ell + 1) = \frac{1}{4}(p^{[n-1]}(\ell) + p^{[n]}(\ell)) + \frac{1}{2}q_{+}.$$

Equivalently, we can write $P(\ell + 1) = A_{(a)} \cdot P(\ell) + b_{(a)}$, where the matrix

$A_{(a)} \in \mathbb{R}^{n \times n}$ and the vector $b_{(a)} \in \mathbb{R}^n$ are given by

$$A_{(a)} = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} & 0 & \dots & \dots & 0 \\ & 1 & \frac{1}{4} & & & \\ & & 1 & \dots & \dots & 0 \\ 1 & 1 & & \dots & \dots & 0 \\ 4 & 2 & 1 & & \dots & 0 \\ 0 & & & & & \end{pmatrix}, \quad b_{(a)} = \begin{pmatrix} \frac{1}{2}q_{-} \\ 0 \\ 2 \\ \vdots \\ \vdots \\ 0 \end{pmatrix}$$

2

$$\begin{matrix} \vdots \\ 4 & 2 \end{matrix}$$

P_* 0 *

Note that the only equilibrium network configuration P_*

respecting the or-

9. References

- Abdallah, C. T. and Tanner, H. G. (2007) Complex networked control systems: introduction to the special section, *IEEE Control Systems Magazine*, 27(4), 30–32.
- Agarwal, P. K. and Sharir, M. (1998) Efficient algorithms for geometric optimization, *ACM Computing Surveys*, 30(4), 412–458.
- Agmon, N. and Peleg, D. (2006) Fault-tolerant gathering algorithms for autonomous mobile robots, *SIAM Journal on Computing*, 36(1), 56–82.
- Alighanbari, M. and How, J. P. (2006) Robust decentralized task assignment for cooperative UAVs, in *AIAA Conference on Guidance, Navigation and Control*, Keystone, CO.
- Ando, H., Oasa, Y., Suzuki, I., and Yamashita, M. (1999) Distributed memoryless point convergence algorithm for mobile robots with limited visibility, *IEEE Transactions on Robotics and Automation*, 15(5), 818–828.
- Angeli, D. (1999) Intrinsic robustness of global asymptotic stability, *Systems & Control Letters*, 38(4-5), 297–307.

- Angeli, D. and Bliman, P.-A. (2006) Stability of leaderless discrete-time multi-agent systems, *Mathematics of Control, Signals and Systems*, 18(4), 293–322.
- Snap-Stabilizing Tasks in Anonymous Networks 955
- Arai, T., Pagello, E., and Parker, L. E. (2002) Guest editorial: Advances in multirobot systems, *IEEE Transactions on Robotics and Automation*, 18(5), 655–661.
- Arkin, R. C. (1998) *Behavior-Based Robotics*, MIT Press, ISBN 0262011654.
- Arsie, A., Savla, K., and Frazzoli, E. (2009) Efficient routing algorithms for multiple vehicles with no explicit communications, *IEEE Transactions on Automatic Control*, to appear.
- Arslan, G., Marden, J. R., and Shamma, J. S. (2007) Autonomous vehicle- target assignment: A game theoretic formulation, *ASME Journal on Dynamic Systems, Measurement, and Control*, 129(5), 584–596.

- Asama, H. (1992) Distributed autonomous robotic system configured with multiple agents and its cooperative behaviors, *Journal of Robotics and Mechatronics*, 4(3), 199–204.
- Aspnes, J., Eren, T., Goldenberg, D. K., Morse, A. S., Whiteley, W., Yang, Y. R., Anderson, B. D. O., and Belhumeur, P. (2006) A theory of network localization, *IEEE Transactions on Mobile Computing*, 5(12), 1663–1678.
- Astolfi, A. (1999) Exponential stabilization of a wheeled mobile robot via discontinuous control, *ASME Journal on Dynamic Systems, Measurement, and Control*, 121(1), 121–127.
- Aurenhammer, F. (1991) Voronoi diagrams: A survey of a fundamental geometric data structure, *ACM Computing Surveys*, 23(3), 345–405.
- Axelrod, R. (1997) The dissemination of culture: A model with local convergence and global polarization, *Journal of Conflict Resolution*, 41(2), 203–226.
- Baillieul, J. and Suri, A. (2003) Information patterns and hedging Brockett's theorem in controlling vehicle formations, in *IEEE Conference on Decision and Control*, pages 556–563, Maui, HI.
- Balch, T. and Parker, L. E., (editors) (2002) *Robot Teams: From Diversity to Polymorphism*, A. K. Peters, ISBN 1568811551.
- Barbašin, E. A. and Krasovskiĭ, N. N. (1952) On stability of motion in the large, *Doklady Akad. Nauk SSSR*, 86, 453–456, (In Russian).
- Barlow, G. W. (1974) Hexagonal territories, *Animal Behavior*, 22, 876–878.
- Barooah, P. (2007) *Estimation and Control with Relative Measurements: Algorithms and Scaling Laws*, Ph.D. thesis, University of California at Santa Barbara.
- Barooah, P. and Hespanha, J. P. (2007) Estimation from relative measurements: Algorithms and scaling laws, *IEEE Control Systems Magazine*, 27(4), 57–74.
- Bartle, R. G. (1995) *The Elements of Integration and Lebesgue Measure*, Wiley-Interscience, ISBN 0471042226.
- Batalin, M. A. and Sukhatme, G. S. (2004) Coverage,

exploration and deployment by a mobile robot and communication network, *Telecommunication Systems Journal*, 26(2), 181–196, Special Issue on Wireless Sensor Networks.

Bauso, D., Giarré, L., and Pesenti, R. (2006) Nonlinear protocols for optimal distributed consensus in networks of dynamic agents, *Systems & Control Letters*, 55(11), 918–928.

- Beard, R. W., McLain, T. W., Goodrich, M. A., and Anderson, E. P. (2002) Coordinated target assignment and intercept for unmanned air vehicles, *IEEE Transactions on Robotics and Automation*, 18(6), 911–922.
- Belta, C. and Kumar, V. (2004) Abstraction and control for groups of robots, *IEEE Transactions on Robotics*, 20(5), 865–875.
- Bertozi, A. L., Kemp, M., and Marthaler, D. (2004) Determining environmental boundaries: Asynchronous communication and physical scales, in *Cooperative Control*, V. Kumar, N. E. Leonard, and A. S. Morse, editors, volume 309 of *Lecture Notes in Control and Information Sciences*, pages 25–42, Springer, ISBN 3540228616.
- Bertsekas, D. P. and Castañón, D. A. (1991) Parallel synchronous and asynchronous implementations of the auction algorithm, *Parallel Computing*, 17, 707–732.
- (1993) Parallel primal-dual methods for the minimum cost flow problem, *Computational Optimization and Applications*, 2(4), 317–336.
- Bertsekas, D. P. and Tsitsiklis, J. N. (1997) *Parallel and Distributed Computation: Numerical Methods*, Athena Scientific, ISBN 1886529019.
- Biggs, N. (1994) *Algebraic Graph Theory*, second edition, Cambridge University Press, ISBN 0521458978.
- Blondel, V. D., Hendrickx, J. M., Olshevsky, A., and Tsitsiklis, J. N. (2005) Convergence in multiagent coordination, consensus, and flocking, in *IEEE Conference on Decision and Control and European Control Conference*, pages 2996–3000, Seville, Spain.
- Boinski, S. and Campbell, A. F. (1995) Use of trill vocalizations to coordinate troop movement among whitefaced capuchins – a 2nd field-test, *Behaviour*, 132, 875–901.
- Boissonnat, J.-D. and Cazals, F. (2002) Smooth surface reconstruction via natural neighbour interpolation of distance functions, *Computational Geometry: Theory and Applications*, 22(1), 185–203.
- Bollobás, B. (2001) *Random Graphs*, second edition, Cambridge University Press, ISBN 0521809207.
- Bollobás, B. and Riordan, O. (2006) *Percolation*,

Cambridge University Press, ISBN 0521872324.

Boltyanski, V., Martini, H., and Soltan, V. (1999) *Geometric methods and optimization problems*, volume 4 of *Combinatorial optimization*, Kluwer Academic Publishers, ISBN 0792354540.

- Boyd, S. (2006) Convex optimization of graph Laplacian eigenvalues, in *Proceedings of the International Congress of Mathematicians*, pages 1311–1319, Madrid, Spain.
- Bruckstein, A. M., Cohen, N., and Efrat, A. (1991) Ants, crickets, and frogs in cyclic pursuit, Technical Report CIS 9105, Center for Intelligent Systems, Technion, Haifa, Israel, available at <http://www.cs.technion.ac.il/tech-reports>.
- Bullo, F., Cortés, J., and Piccoli, B. (2009) Special issue on control and optimization in cooperative networks, *SIAM Journal on Control and Optimization*, 48(1), vii–vii.
- Bullo, F. and Lewis, A. D. (2004) *Geometric Control of Mechanical Systems*, volume 49 of *Texts in Applied Mathematics*, Springer, ISBN 0387221956.
- Bulusu, N., Heidemann, J., and Estrin, D. (2001) Adaptive beacon placement, in *International Conference on Distributed Computing Systems*, pages 489–498, Mesa, AZ.
- Burgard, W., Moors, M., Stachniss, C., and Schneider, F. E. (2005) Coordinated multi-robot exploration, *IEEE Transactions on Robotics*, 21(3), 376–386.
- Caicedo-Núñez, C. H. and Žefran, M. (2008) Performing coverage on nonconvex domains, in *IEEE Conference on Control Applications*, pages 1019–1024, San Antonio, TX.
- Cao, M. (2007) *Multi-Agent Formations and Sensor Networks*, Ph.D. thesis, Yale University.
- Cao, M., Morse, A. S., and Anderson, B. D. O. (2008) Reaching a consensus in a dynamically changing environment - convergence rates, measurement delays and asynchronous events, *SIAM Journal on Control and Optimization*, 47(2), 601–623.
- Cao, Y. U., Fukunaga, A. S., and Kahng, A. (1997) Cooperative mobile robotics: Antecedents and directions, *Autonomous Robots*, 4(1), 7–27.
- Carli, R. (2008) *Topics in the Average Consensus Problems*, Ph.D. thesis, Università di Padova, Italy.

- Carli, R. and Bullo, F. (2009) Quantized coordination algorithms for rendezvous and deployment, *SIAM Journal on Control and Optimization*, (Submitted Dec. 2007) to appear.
- Carli, R., Bullo, F., and Zampieri, S. (2009) Quantized average consensus via dynamic coding/decoding schemes, *International Journal on Robust and Nonlinear Control*, (Submitted May 2008) to appear.

- Carli, R., Fagnani, F., Speranzon, A., and Zampieri, S. (2008) Communication constraints in the average consensus problem, *Automatica*, 44(3), 671–684.
- Casbeer, D. W., Kingston, D. B., Beard, R. W., McLain, T. W., Li, S.-M., and Mehra, R. (2006) Cooperative forest fire surveillance using a team of small unmanned air vehicles, *International Journal of Systems Sciences*, 37(6), 351–360.
- Casbeer, D. W., Li, S.-M., Beard, R. W., Mehra, R. K., and McLain, T. W. (2005) Forest fire monitoring with multiple small UAVs, in *American Control Conference*, pages 3530–3535, Portland, OR.
- Cassandras, C. G. and Lafontaine, S. (2007) *Introduction to Discrete-Event Systems*, second edition, Springer, ISBN 0387333320.
- Castañón, D. A. and Wu, C. (2003) Distributed algorithms for dynamic reassignment, in *IEEE Conference on Decision and Control*, pages 13–18, Maui, HI.
- Chatterjee, S. and Seneta, E. (1977) Towards consensus: Some convergence theorems on repeated averaging, *Journal of Applied Probability*, 14(1), 89–97.
- Chavel, I. (1984) *Eigenvalues in Riemannian Geometry*, Academic Press, ISBN 0121706400.
- Chen, C.-T. (1984) *Linear System Theory and Design*, Holt, Rinehart, and Winston, ISBN 0030602890.
- Chopra, N. and Spong, M. W. (2009) On exponential synchronization of Kuramoto oscillators, *IEEE Transactions on Automatic Control*, 54(2), 353–357.
- Chorin, A. J. and Marsden, J. E. (1994) *A Mathematical Introduction to Fluid Mechanics*, volume 4 of *Texts in Applied Mathematics*, third edition, Springer, ISBN 0387979182.
- Choset, H. (2001) Coverage for robotics – A survey of recent results, *Annals of Mathematics and Artificial Intelligence*, 31(1-4), 113–126.
- Chvátal, V. (1975) A combinatorial theorem in plane geometry, *Journal of Combinatorial Theory. Series B*, 18, 39–41.
- Clark, J. and Fierro, R. (2007) Mobile robotic sensors for

perimeter detection and tracking, *ISA Transactions*,
46(1), 3–13.

Clarke, F. H. (1983) *Optimization and Nonsmooth Analysis*, Canadian
Mathematical Society Series of Monographs and Advanced Texts, John Wiley,
ISBN 047187504X.

- Cogburn, R. (1984) The ergodic theory of Markov chains in random environments, *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*, 66(1), 109–128.
- Conradt, L. and Roper, T. J. (2003) Group decision-making in animals, *Nature*, 421(6919), 155–158.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001) *Introduction to Algorithms*, second edition, MIT Press, ISBN 0262032937.
- Cortés, J. (2006) Finite-time convergent gradient flows with applications to network consensus, *Automatica*, 42(11), 1993–2000.
- (2007) Distributed Kriged Kalman filter for spatial estimation, *IEEE Transactions on Automatic Control*, submitted.
- (2008a) Discontinuous dynamical systems – a tutorial on solutions, non-smooth analysis, and stability, *IEEE Control Systems Magazine*, 28(3), 36–73.
- (2008b) Distributed algorithms for reaching consensus on general functions, *Automatica*, 44(3), 726–737.
- Cortés, J. and Bullo, F. (2005) Coordination and geometric optimization via distributed dynamical systems, *SIAM Journal on Control and Optimization*, 44(5), 1543–1574.
- Cortés, J., Martínez, S., and Bullo, F. (2005) Spatially-distributed coverage optimization and control with limited-range interactions, *ESAIM: Control, Optimisation & Calculus of Variations*, 11, 691–719.
- (2006) Robust rendezvous for mobile autonomous agents via proximity graphs in arbitrary dimensions, *IEEE Transactions on Automatic Control*, 51(8), 1289–1298.
- Cortés, J., Martínez, S., Karatas, T., and Bullo, F. (2004) Coverage control for mobile sensing networks, *IEEE Transactions on Robotics and Automation*, 20(2), 243–255.
- Couzin, I. D., Krause, J., Franks, N. R., and Levin, S. A. (2005) Effective leadership and decision-making in animal groups on the move, *Nature*, 433(7025), 513–516.

- Cucker, F. and Smale, S. (2007) Emergent behavior in flocks, *IEEE Transactions on Automatic Control*, 52(5), 852–862.
- Cybenko, G. (1989) Dynamic load balancing for distributed memory multiprocessors, *Journal of Parallel and Distributed Computing*, 7(2), 279–301.

- de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (2000) *Computational Geometry: Algorithms and Applications*, second edition, Springer, ISBN 3540656200.
- de Gennaro, M. C. and Jadbabaie, A. (2006) Decentralized control of connectivity for multi-agent systems, in *IEEE Conference on Decision and Control*, pages 3628–3633, San Diego, CA.
- de Silva, V. and Ghrist, R. (2007) Coverage in sensor networks via persistent homology, *Algebraic & Geometric Topology*, 7, 339–358.
- Deffuant, G., Neau, D., Amblard, F., and Weisbuch, G. (2000) Mixing beliefs among interacting agents, *Advances in Complex Systems*, 3(1/4), 87–98.
- DeGroot, M. H. (1974) Reaching a consensus, *Journal of the American Statistical Association*, 69(345), 118–121.
- Dias, M. B., Zlot, R., Kalra, N., and Stentz, A. (2006) Market-based multi-robot coordination: A survey and analysis, *Proceedings of the IEEE*, 94(7), 1257–1270.
- Diestel, R. (2005) *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*, second edition, Springer, ISBN 978-3-540-26182-7.
- Dimarogonas, D. V. and Kyriakopoulos, K. J. (2007) On the rendezvous problem for multiple nonholonomic agents, *IEEE Transactions on Automatic Control*, 52(5), 916–922.
- Drezner, Z., (editor) (1995) *Facility Location: A Survey of Applications and Methods*, Series in Operations Research, Springer, ISBN 0-387-94545-8.
- Drezner, Z. and Hamacher, H. W., (editors) (2001) *Facility Location: Applications and Theory*, Springer, ISBN 3540421726.
- Du, Q., Faber, V., and Gunzburger, M. (1999) Centroidal Voronoi tessellations: Applications and algorithms, *SIAM Review*, 41(4), 637–676.
- Dubins, L. E. (1957) On curves of minimal length with a constraint on average curvature and with prescribed initial and terminal positions and tangents, *American Journal of Mathematics*, 79, 497–516.

Dullerud, G. E. and Paganini, F. (2000) *A Course in Robust Control Theory*, number 36 in Texts in Applied Mathematics, Springer, ISBN 978-0-387- 98945-7.

Dunbar, W. B. and Murray, R. M. (2006) Distributed receding horizon control for multi-vehicle formation stabilization, *Automatica*, 42(4), 549–558.

Durrett, R. (2006) *Random Graph Dynamics*, Series in Statistical and Probabilistic Mathematics, Cambridge University Press, ISBN 0521866561.

- Dynia, M., Kutylowski, J., Meyer auf der Heide, F., and Schindelhauer, C. (2006) Smart robot teams exploring sparse trees, in *International Symposium of Mathematical Foundations of Computer Science*, Stará Lesná, Slovakia.
- Efrat, A. and Har-Peled, S. (2006) Guarding galleries and terrains, *Information Processing Letters*, 100(6), 238–245.
- Eidenbenz, S., Stamm, C., and Widmayer, P. (2001) Inapproximability results for guarding polygons and terrains, *Algorithmica*, 31(1), 79–113.
- Emerson, A. E. (1994) Temporal and modal logic, in *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, J. van Leeuwen, editor, pages 997–1072, MIT Press, ISBN 0262720159.
- Fagnani, F., Johansson, K. H., Speranzon, A., and Zampieri, S. (2004) On multi-vehicle rendezvous under quantized communication, in *Mathematical Theory of Networks and Systems*, Leuven, Belgium, Electronic Proceedings.
- Fagnani, F. and Zampieri, S. (2009) Average consensus with packet drop communication, *SIAM Journal on Control and Optimization*, 48(1), 102–133.
- Fainekos, G. E., Kress-Gazit, H., and Pappas, G. J. (2005) Temporal logic motion planning for mobile robots, in *IEEE International Conference on Robotics and Automation*, pages 2032–2037, Barcelona, Spain.
- Fang, L. and Antsaklis, P. J. (2008) Asynchronous consensus protocols using nonlinear paracontractions theory, *IEEE Transactions on Automatic Control*, 53, 2351–2355.
- Fekete, S. P., Mitchell, J. S. B., and Beurer, K. (2005) On the continuous Fermat–Weber problem, *Operations Research*, 53(1), 61 – 76.
- Ferrari-Trecate, G., Buffa, A., and Gati, M. (2006) Analysis of coordination in multi-agent systems through partial difference equations, *IEEE Transactions on Automatic Control*, 51(6), 1058–1063.
- Fiedler, M. (1986) *Special Matrices and their Applications in Numerical Mathematics*, Martinus Nijhoff Publishers, ISBN 9024729572.
- Flocchini, P., Prencipe, G., Santoro, N., and Widmayer, P. (1999) Hard tasks for weak robots: The role of common knowledge in pattern formation by

autonomous mobile robots, in *ISAAC 1999, 10th International Symposium on Algorithm and Computation (Chennai, India)*, A. Aggarwal and C. P. Rangan, editors, volume 1741 of *Lecture Notes in Computer Science*, pages 93–102, Springer, ISBN 3540669167.

— (2005) Gathering of asynchronous oblivious robots with limited visibility,

Theoretical Computer Science, 337(1-3), 147–168.

Fraigniaud, P., Gąsieniec, L., Kowalski, D. R., and Pelc, A. (2004) Collective tree exploration, in *LATIN 2004: Theoretical Informatics*, M. Farach-Colton, editor, volume 2976 of *Lecture Notes in Computer Science*, pages 141–151, Springer, ISBN 3540212582.

Francis, B. A. (2006) Distributed control of autonomous mobile robots, course Notes, Version 1.01, University of Toronto, Canada.

Frazzoli, E. and Bullo, F. (2004) Decentralized algorithms for vehicle routing in a stochastic time-varying environment, in *IEEE Conference on Decision and Control*, pages 3357–3363, Paradise Island, Bahamas.

Gallager, R. G. (1968) *Information Theory and Reliable Communication*, John Wiley, ISBN 0471290483.

Gallager, R. G., Humblet, P. A., and Spira, P. M. (1983) A distributed algorithm for minimum-weight spanning trees, *ACM Transactions on Programming Languages and Systems*, 5(1), 66–77.

Ganguli, A., Cortés, J., and Bullo, F. (2007) Distributed coverage of nonconvex environments, in *Networked Sensing Information and Control (Proceedings of the NSF Workshop on Future Directions in Systems Research for Networked Sensing, May 2006, Boston, MA)*, V. Saligrama, editor, pages 289–305, Lecture Notes in Control and Information Sciences, Springer, ISBN 0387688439.

— (2009) Multirobot rendezvous with visibility sensors in nonconvex environments, *IEEE Transactions on Robotics*, 25(2), (Submitted Nov. 2006) to appear.

Gao, C., Cortés, J., and Bullo, F. (2008) Notes on averaging over acyclic digraphs and discrete coverage control, *Automatica*, 44(8), 2120–2127.

Gazi, V. and Passino, K. M. (2003) Stability analysis of swarms, *IEEE Transactions on Automatic Control*, 48(4), 692–697.

- Gerkey, B. P. and Mataric, M. J. (2004) A formal analysis and taxonomy of task allocation in multi-robot systems, *International Journal of Robotics Research*, 23(9), 939–954.
- Ghosh, S. K. (1987) Approximation algorithms for Art Gallery Problems, in *Proceedings of the Canadian Information Processing Society*, pages 429–434.
- Godsil, C. D. and Royle, G. F. (2001) *Algebraic Graph Theory*, volume 207 of *Graduate Texts in Mathematics*, Springer, ISBN 0387952411.

- Godwin, M. F., Spry, S., and Hedrick, J. K. (2006) Distributed collaboration with limited communication using mission state estimates, in *American Control Conference*, pages 2040–2046, Minneapolis, MN.
- Goebel, R., Hespanha, J. P., Teel, A. R., Cai, C., and Sanfelice, R. G. (2004) Hybrid systems: generalized solutions and robust stability, in *IFAC Symposium on Nonlinear Control Systems*, pages 1–12, Stuttgart, Germany.
- Goodman, J. E. and O'Rourke, J., (editors) (2004) *Handbook of Discrete and Computational Geometry*, second edition, CRC Press, ISBN 1584883014.
- Graham, R. and Cortés, J. (2009) Asymptotic optimality of multicenter Voronoi configurations for random field estimation, *IEEE Transactions on Automatic Control*, 54(1), 153–158.
- Gray, R. M. and Neuhoff, D. L. (1998) Quantization, *IEEE Transactions on Information Theory*, 44(6), 2325–2383, Commemorative Issue 1948-1998.
- Gruber, P. M. (1983) Approximation of convex bodies, in *Convexity and its Applications*, P. M. Gruber and J. M. Willis, editors, pages 131–162, Birkhäuser, ISBN 3764313846.
- Guckenheimer, J. and Holmes, P. (1990) *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*, volume 42 of *Applied Mathematical Sciences*, Springer, ISBN 0387908196.
- Gueron, S. and Levin, S. A. (1993) Self-organization of front patterns in large wildebeest herds, *Journal of Theoretical Biology*, 165, 541–552.
- Gupta, P. and Kumar, P. R. (2000) The capacity of wireless networks, *IEEE Transactions on Information Theory*, 46(2), 388–404.
- Gupta, V., Chung, T. H., Hassibi, B., and Murray, R. M. (2006a) On a stochastic sensor selection algorithm with applications in sensor scheduling and sensor coverage, *Automatica*, 42(2), 251–260.
- Gupta, V., Langbort, C., and Murray, R. M. (2006b) On the robustness of distributed algorithms, in *IEEE Conference on Decision and Control*, pages 3473–3478, San Diego, CA.
- Hatano, Y. and Mesbahi, M. (2005) Agreement over random networks, *IEEE Transactions on Automatic Control*, 50(11), 1867–1872.

Hayes, J., McJunkin, M., and Košěcká, J. (2003) Communication enhanced navigation strategies for teams of mobile agents, in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pages 2285–2290, Las Vegas, NV.

- Hegselmann, R. and Krause, U. (2002) Opinion dynamics and bounded confidence models, analysis, and simulations, *Journal of Artificial Societies and Social Simulation*, 5(3).
- Hendrickx, J. M. (2008) *Graphs and Networks for the Analysis of Autonomous Agent Systems*, Ph.D. thesis, Université Catholique de Louvain, Belgium.
- Hendrickx, J. M., Anderson, B. D. O., Delvenne, J.-C., and Blondel, V. D. (2007) Directed graphs for the analysis of rigidity and persistence in autonomous agents systems, *International Journal on Robust and Nonlinear Control*, 17(10), 960–981.
- Hernández-Peñalver, G. (1994) Controlling guards, in *Canadian Conference on Computational Geometry*, pages 387–392, Saskatoon, Canada.
- Hibbeler, R. C. (2006) *Engineering Mechanics: Statics & Dynamics*, 11th edition, Prentice Hall, ISBN 0132215098.
- Horn, R. A. and Johnson, C. R. (1985) *Matrix Analysis*, Cambridge University Press, ISBN 0521386322.
- Howard, A., Mataric, M. J., and Sukhatme, G. S. (2002) Mobile sensor network deployment using potential fields: A distributed scalable solution to the area coverage problem, in *International Conference on Distributed Autonomous Robotic Systems*, pages 299–308, Fukuoka, Japan.
- Howard, A., Parker, L. E., and Sukhatme, G. S. (2006) Experiments with a large heterogeneous mobile robot team: Exploration, mapping, deployment, and detection, *International Journal of Robotics Research*, 25(5-6), 431–447.
- Hu, J., Prandini, M., and Tomlin, C. (2007) Conjugate points in formation constrained optimal multi-agent coordination: A case study, *SIAM Journal on Control and Optimization*, 45(6), 2119–2137.
- Hussein, I. I. and Stipanović, D. M. (2007) Effective coverage control for mobile sensor networks with guaranteed collision avoidance, *IEEE Transactions on Control Systems Technology*, 15(4), 642–657.
- Huygens, C. (1673) *Horologium Oscillatorium*, Paris, France.
- Igarashi, Y., Hatanaka, T., Fujita, M., and Spong, M. W. (2007) Passivity-based 3D attitude coordination: Convergence and connectivity, in *IEEE Conference on*

Decision and Control, pages 2558–2565, New Orleans, LA.

Jadbabaie, A., Lin, J., and Morse, A. S. (2003) Coordination of groups of mobile autonomous agents using nearest neighbor rules, *IEEE Transactions on Automatic Control*, 48(6), 988–1001.

- Jadbabaie, A., Motee, N., and Barahona, M. (2004) On the stability of the Kuramoto model of coupled nonlinear oscillators, in *American Control Conference*, pages 4296–4301, Boston, MA.
- Jaromczyk, J. W. and Toussaint, G. T. (1992) Relative neighborhood graphs and their relatives, *Proceedings of the IEEE*, 80(9), 1502–1517.
- Ji, M. and Egerstedt, M. (2007) Distributed control of multiagent systems while preserving connectedness, *IEEE Transactions on Robotics*, 23(4), 693–703.
- Jiang, Z.-P. and Wang, Y. (2001) Input-to-state stability for discrete-time nonlinear systems, *Automatica*, 37(6), 857–869.
- Johansson, K. J., Egerstedt, M., Lygeros, J., and Sastry, S. S. (1999) On the regularization of Zeno hybrid automata, *Systems & Control Letters*, 38(3), 141–150.
- Justh, E. W. and Krishnaprasad, P. S. (2004) Equilibria and steering laws for planar formations, *Systems & Control Letters*, 52(1), 25–38.
- (2006) Steering laws for motion camouflage, *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 462(2076), 3629–3643.
- Kang, K., Yan, J., and Bitmead, R. R. (2006) Communication resources for disturbance rejection in coordinated vehicle control, in *IEEE Conference on Decision and Control and European Control Conference*, pages 5730–5735, Seville, Spain.
- Kashyap, A., Başar, T., and Srikant, R. (2007) Quantized consensus, *Automatica*, 43(7), 1192–1203.
- Khalil, H. K. (2002) *Nonlinear Systems*, third edition, Prentice Hall, ISBN 0130673897.
- Kim, Y. and Mesbahi, M. (2006) On maximizing the second smallest eigenvalue of a state-dependent graph Laplacian, *IEEE Transactions on Automatic Control*, 51(1), 116–120.

Klavins, E. (2003) Communication complexity of multi-robot systems, in

Algorithmic Foundations of Robotics V, J.-D. Boissonnat, J. W. Burdick,

K. Goldberg, and S. Hutchinson, editors, volume 7 of *Tracts in Advanced Robotics*, Springer, Berlin Heidelberg, ISBN 3540404767.

Klavins, E., Ghrist, R., and Lipsky, D. (2006) A grammatical approach to self-organizing robotic systems, *IEEE Transactions on Automatic Control*, 51(6), 949–962.

- Klavins, E. and Murray, R. M. (2004) Distributed algorithms for cooperative control, *IEEE Pervasive Computing*, 3(1), 56–65.
- Korte, B. and Vygen, J. (2005) *Combinatorial Optimization: Theory and Algorithms*, volume 21 of *Algorithmics and Combinatorics*, third edition, Springer, ISBN 3540256849.
- Krasovskiĭ, N. N. (1963) *Stability of motion. Applications of Lyapunov's second method to differential systems and equations with delay*, Stanford University Press, translated by J. L. Brenner.
- Krick, L. (2007) *Application of Graph Rigidity in Formation Control of Multi-Robot Networks*, Master's thesis, University of Toronto, Canada.
- Kuramoto, Y. (1975) Self-entrainment of a population of coupled non-linear oscillators, in *International Symposium on Mathematical Problems in Theoretical Physics*, H. Araki, editor, volume 39 of *Lecture Notes in Physics*, pages 420–422, Springer, ISBN 978-3-540-07174-7.
- (1984) *Chemical oscillations, waves, and turbulence*, Springer, ISBN 0387133224.
- Kwok, A. and Martínez, S. (2009) Deployment algorithms for a power-constrained mobile sensor network, *International Journal on Robust and Nonlinear Control*, to appear.
- Lafferriere, G., Williams, A., Caughman, J., and Veerman, J. J. P. (2005) Decentralized control of vehicle formations, *Systems & Control Letters*, 54(9), 899–910.
- Landau, H. J. and Odlyzko, A. M. (1981) Bounds for eigenvalues of certain stochastic matrices, *Linear Algebra and its Applications*, 38, 5–15.
- Langbort, C. and Gupta, V. (2009) Minimal interconnection topology in distributed control, *SIAM Journal on Control and Optimization*, 48(1), 397–413.
- Langetepe, E. and Zachmann, G. (2006) *Geometric Data Structures for Computer Graphics*, A. K. Peters, ISBN 1568812353.
- Lanthier, M., Nussbaum, D., and Wang, T.-J. (2005) Calculating the meeting point of scattered robots on weighted terrain surfaces, in *Computing: The Australasian Theory Symposium (CATS)*, volume 27, pages 107–118, Newcastle, Australia.

- LaSalle, J. P. (1960) Some extensions of Liapunov's second method, *IRE Trans. Circuit Theory*, CT-7, 520–527.
- (1986) *The Stability and Control of Discrete Processes*, volume 62 of *Applied Mathematical Sciences*, Springer, ISBN 0387964118.

- Laventall, K. and Cortés, J. (2009) Coverage control by robotic networks with limited-range anisotropic sensory, *International Journal of Control*, 82, to appear.
- Lee, D. and Spong, M. W. (2007) Stable flocking of multiple inertial agents on balanced graphs, *IEEE Transactions on Automatic Control*, 52(8), 1469–1475.
- Lee, D. T. and Lin, A. K. (1986) Computational complexity of art gallery problems, *IEEE Transactions on Information Theory*, 32(2), 276–282.
- Lekien, F. and Leonard, N. E. (2009) Non-uniform coverage and cartograms, *SIAM Journal on Control and Optimization*, 48(1), 351–372.
- Li, X.-Y. (2003) Algorithmic, geometric and graphs issues in wireless networks, *Wireless Communications and Mobile Computing*, 3(2), 119–140.
- Lin, J., Morse, A. S., and Anderson, B. D. O. (2007a) The multi-agent rendezvous problem. Part 1: The synchronous case, *SIAM Journal on Control and Optimization*, 46(6), 2096–2119.
- (2007b) The multi-agent rendezvous problem. Part 2: The asynchronous case, *SIAM Journal on Control and Optimization*, 46(6), 2120–2147.
- Lin, Z. (2005) *Coupled Dynamic Systems: From Structure Towards Stability and Stabilizability*, Ph.D. thesis, University of Toronto, Canada.
- Lin, Z., Broucke, M., and Francis, B. (2004) Local control strategies for groups of mobile autonomous agents, *IEEE Transactions on Automatic Control*, 49(4), 622–629.
- Lin, Z., Francis, B., and Maggiore, M. (2005) Necessary and sufficient graphical conditions for formation control of unicycles, *IEEE Transactions on Automatic Control*, 50(1), 121–127.
- (2007c) State agreement for continuous-time coupled nonlinear systems, *SIAM Journal on Control and Optimization*, 46(1), 288–307.
- Lloyd, E. L., Liu, R., Marathe, M. V., Ramanathan, R., and Ravi, S. S. (2005) Algorithmic aspects of topology control problems for ad hoc networks, *Mobile Networks and Applications*, 10(1-2), 19–34.
- Lorenz, J. (2007) *Repeated Averaging and Bounded Confidence – Modeling, Analysis and Simulation of Continuous Opinion Dynam-*

ics, Ph.D. thesis, University of Bremen, Germany, available at <http://nbn-resolving.de/urn:nbn:de:gbv:46-diss000106688>.

Lorenz, J. and Lorenz, D. A. (2008) On conditions for convergence to consensus, available at <http://arxiv.org/abs/0803.2211>.

- Lovász, L. (1993) Random walks on graphs: A survey, in *Combinatorics: Paul Erdős is Eighty*, T. S. D. Miklós, V. T. Sós, editor, volume 2, pages 353–398, János Bolyai Mathematical Society, ISBN 9638022744.
- Lumelsky, V. J. and Harinarayan, K. R. (1997) Decentralized motion planning for multiple mobile robots: The cocktail party model, *Autonomous Robots*, 4(1), 121–135.
- Lygeros, J., Johansson, K. H., Simić, S. N., Zhang, J., and Sastry, S. S. (2003) Dynamical properties of hybrid automata, *IEEE Transactions on Automatic Control*, 48(1), 2–17.
- Lynch, N. A. (1997) *Distributed Algorithms*, Morgan Kaufmann, ISBN 1558603484.
- Lynch, N. A., Segala, R., and Vaandrager, F. (2003) Hybrid I/O automata, *Information and Computation*, 185(1), 105–157.
- Marshall, J. A., Broucke, M. E., and Francis, B. A. (2004) Formations of vehicles in cyclic pursuit, *IEEE Transactions on Automatic Control*, 49(11), 1963–1974.
- Marthaler, D. and Bertozzi, A. L. (2003) Tracking environmental level sets with autonomous vehicles, in *Recent Developments in Cooperative Control and Optimization*, S. Butenko, R. Murphey, and P. M. Pardalos, editors, pages 317–330, Kluwer Academic Publishers, ISBN 1402076444.
- Martínez, S. (2009a) Distributed interpolation schemes for field estimation by mobile sensor networks, *IEEE Transactions on Control Systems Technology*, to appear.
- (2009b) Practical multiagent rendezvous through modified circumcenter algorithms, *Automatica*, to appear.
- Martínez, S. and Bullo, F. (2006) Optimal sensor placement and motion coordination for target tracking, *Automatica*, 42(4), 661–668.
- Martínez, S., Bullo, F., Cortés, J., and Frazzoli, E. (2007a) On synchronous robotic networks – Part I: Models, tasks and complexity, *IEEE Transactions on Automatic Control*, 52(12), 2199–2213.
- (2007b) On synchronous robotic networks – Part II: Time complexity of rendezvous and deployment algorithms, *IEEE Transactions on Automatic*

Control, 52(12), 2214–2226.

Martínez, S., Cortés, J., and Bullo, F. (2007c) Motion coordination with distributed information, *IEEE Control Systems Magazine*, 27(4), 75–88.

- McLure, D. E. and Vitale, R. A. (1975) Polygonal approximation of plane convex bodies, *Journal of Mathematical Analysis and Applications*, 51(2), 326–358.
- Meester, R. and Roy, R. (2008) *Continuum Percolation*, Cambridge University Press, ISBN 0521062500.
- Mehaute, A. L., Laurent, P. J., and Schumaker, L. L., (editors) (1993) *Curves and Surfaces in Geometric Design*, A. K. Peters, ISBN 1568810393.
- Merris, R. (1994) Laplacian matrices of a graph: A survey, *Linear Algebra its Applications*, 197, 143–176.
- Mesbahi, M. (2005) On state-dependent dynamic graphs and their controllability properties, *IEEE Transactions on Automatic Control*, 50(3), 387–392.
- Meyer, C. D. (2001) *Matrix Analysis and Applied Linear Algebra*, SIAM, ISBN 0898714540.
- Meyn, S. and Tweedie, R. (1999) *Markov Chains and Stochastic Stability*, Springer, ISBN 3540198326.
- Miller, M. B. and Bassler, B. L. (2001) Quorum sensing in bacteria, *Annual Review of Microbiology*, 55, 165–199.
- Mirollo, R. E. and Strogatz, S. H. (1990) Synchronization of pulse-coupled biological oscillators, *SIAM Journal on Applied Mathematics*, 50(6), 1645–1662.
- Mitchell, J. S. B. (1997) Shortest paths and networks, in *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, editors, chapter 24, pages 445–466, CRC Press, ISBN 0849385245.
- Mohar, B. (1991) The Laplacian spectrum of graphs, in *Graph Theory, Combinatorics, and Applications*, Y. Alavi, G. Chartrand, O. R. Oellermann, and A. J. Schwenk, editors, volume 2, pages 871–898, John Wiley, ISBN 0471532452.
- Moore, B. J. and Passino, K. M. (2007) Distributed task assignment for mobile agents, *IEEE Transactions on Automatic Control*, 52(4), 749–753.
- Moreau, L. (2003) Time-dependent unidirectional communication in multi-agent systems, available at <http://arxiv.org/abs/math/0306426>.
- (2004) Stability of continuous-time distributed consensus algorithms, available at <http://arxiv.org/abs/math/0409010>.

289

— (2005) Stability of multiagent systems with time-dependent communication links, *IEEE Transactions on Automatic Control*, 50(2), 169–182.

Moses, Y. and Tennenholtz, M. (1995) Artificial social systems, *Computers and AI*, 14(6), 533–562.

Moshtagh, N. and Jadbabaie, A. (2007) Distributed geodesic control laws for flocking of nonholonomic agents, *IEEE Transactions on Automatic Control*, 52(4), 681–686.

Nijmeijer, H. (2001) A dynamical control view on synchronization, *Physica D*, 154(3-4), 219–228.

Ögren, P., Fiorelli, E., and Leonard, N. E. (2004) Cooperative control of mobile sensor networks: Adaptive gradient climbing in a distributed environment, *IEEE Transactions on Automatic Control*, 49(8), 1292–1302.

Oh, S., Schenato, L., Chen, P., and Sastry, S. S. (2007) Tracking and coordination of multiple agents using sensor networks: system design, algorithms and experiments, *Proceedings of the IEEE*, 95(1), 163–187.

Okabe, A., Boots, B., Sugihara, K., and Chiu, S. N. (2000) *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, second edition, Wiley Series in Probability and Statistics, John Wiley, ISBN 0471986356.

Okubo, A. (1986) Dynamical aspects of animal grouping: swarms, schools, flocks and herds, *Advances in Biophysics*, 22, 1–94.

Olfati-Saber, R. (2005) Ultrafast consensus in small world networks, in *American Control Conference*, pages 2371–2378, Portland, OR.

— (2006) Flocking for multi-agent dynamic systems: Algorithms and theory, *IEEE Transactions on Automatic Control*, 51(3), 401–420.

Olfati-Saber, R., Fax, J. A., and Murray, R. M. (2007) Consensus and cooperation in networked multi-agent systems, *Proceedings of the IEEE*, 95(1), 215–233.

- Olfati-Saber, R., Franco, E., Frazzoli, E., and Shamma, J. S. (2006) Belief consensus and distributed hypothesis testing in sensor networks, in *Network Embedded Sensing and Control. (Proceedings of NESC'05 Worskhop)*, P. Antsaklis and P. Tabuada, editors, volume 331 of *Lecture Notes in Control and Information Sciences*, pages 169–182, Springer, ISBN 3540327940.
- Olfati-Saber, R. and Murray, R. M. (2002) Graph rigidity and distributed formation stabilization of multi-vehicle systems, in *IEEE Conference on Decision and Control*, pages 2965–2971, Las Vegas, NV.

— (2004) Consensus problems in networks of agents with switching topology and time-delays, *IEEE Transactions on Automatic Control*, 49(9), 1520–1533.

Olshevsky, A. and Tsitsiklis, J. N. (2009) Convergence speed in distributed consensus and averaging, *SIAM Journal on Control and Optimization*, 48(1), 33–55.

O’Rourke, J. (2000) *Computational Geometry in C*, Cambridge University Press, ISBN 0521649765.

Paley, D. A., Leonard, N. E., Sepulchre, R., Grunbaum, D., and Parrish,

J. K. (2007) Oscillator models and collective motion, *IEEE Control Systems Magazine*, 27(4), 89–105.

Pallottino, L., Scordio, V. G., Frazzoli, E., and Bicchi, A. (2007) Decentralized cooperative policy for conflict resolution in multi-vehicle systems, *IEEE Transactions on Robotics*, 23(6), 1170–1183.

Papachristodoulou, A. and Jadbabaie, A. (2006) Synchronization in oscillator networks with heterogeneous delays, switching topologies and nonlinear dynamics, in *IEEE Conference on Decision and Control*, pages 4307–4312, San Diego, CA.

Parhami, B. (1999) *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum Series in Computer Science, Springer, ISBN 0306459701.

Parrish, J. K., Viscido, S. V., and Grunbaum, D. (2002) Self-organized fish schools: an examination of emergent properties, *Biological Bulletin*, 202, 296–305.

Passino, K. M. (2004) *Biomimicry for Optimization, Control, and Automation*, Springer, ISBN 1852338040.

Patterson, S., Bamieh, B., and Abbadi, A. E. (2007) Distributed average consensus with stochastic communication failures, in *IEEE Conference on Decision and Control*, pages 4215–4220, New Orleans, LA.

- Pavone, M. and Frazzoli, E. (2007) Decentralized policies for geometric pattern formation and path coverage, *ASME Journal on Dynamic Systems, Measurement, and Control*, 129(5), 633–643.
- Pavone, M., Frazzoli, E., and Bullo, F. (2007) Decentralized algorithms for stochastic and dynamic vehicle routing with general target distribution, in *IEEE Conference on Decision and Control*, pages 4869–4874, New Orleans, LA.
- Payton, D., Daily, M., Estowski, R., Howard, M., and Lee, C. (2001) Pheromone robotics, *Autonomous Robots*, 11(3), 319–324.

- Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, ISBN 1558604790.
- Peleg, D. (2000) *Distributed Computing. A Locality-Sensitive Approach*, Monographs on Discrete Mathematics and Applications, SIAM, ISBN 0898714648.
- Penrose, M. (2003) *Random Geometric Graphs*, Oxford Studies in Probability, Oxford University Press, ISBN 0198506260.
- Picci, G. and Taylor, T. (2007) Almost sure convergence of random gossip algorithms, in *IEEE Conference on Decision and Control*, pages 282–287, New Orleans, LA.
- Pimenta, L. C. A., Kumar, V., Mesquita, R. C., and Pereira, G. A. S. (2008) Sensing and coverage for a network of heterogeneous robots, in *IEEE Conference on Decision and Control*, pages 3947–3952, Cancún, México.
- Pinciu, V. (2003) A coloring algorithm for finding connected guards in art galleries, in *Discrete Mathematical and Theoretical Computer Science*, volume 2731/2003 of *Lecture Notes in Computer Science*, pages 257–264, Springer.
- Poduri, S. and Sukhatme, G. S. (2004) Constrained coverage for mobile sensor networks, in *IEEE International Conference on Robotics and Automation*, pages 165–172, New Orleans, LA.
- Porfiri, M. and Stilwell, D. J. (2007) Consensus seeking over random weighted directed graphs, *IEEE Transactions on Automatic Control*, 52(9), 1767–1773.
- Preparata, F. P. and Shamos, M. I. (1993) *Computational Geometry: An Introduction*, Springer, ISBN 0387961313.
- Radke, J. D. (1988) On the shape of a set of points, in *Computational morphology. A computational geometric approach to the analysis of form.*, G. T. Toussaint, editor, pages 105–136, North-Holland, ISBN 0-444-70467-1.
- Rao, B. S. Y. and Durrant-Whyte, H. F. (1993) A decentralized Bayesian algorithm for identification of tracked targets, *IEEE Transactions on Systems, Man, & Cybernetics*, 23(6), 1683–1698.

Rathinam, S., Sengupta, R., and Darbha, S. (2007) A resource allocation algorithm for multi-vehicle systems with non holonomic constraints, *IEEE Transactions on Automation Sciences and Engineering*, 4(1), 98–104.

- Reeds, J. A. and Shepp, L. A. (1990) Optimal paths for a car that goes both forwards and backwards, *Pacific Journal of Mathematics*, 145(2), 367–393.
- Rekleitis, I. M., Dudek, G., and Milios, E. (2001) Multi-robot collaboration for robust exploration, *Annals of Mathematics and Artificial Intelligence*, 31(1-4), 7–40.
- Ren, W. and Beard, R. W. (2005) Consensus seeking in multi-agent systems under dynamically changing interaction topologies, *IEEE Transactions on Automatic Control*, 50(5), 655–661.
- (2008) *Distributed Consensus in Multi-vehicle Cooperative Control*, Communications and Control Engineering, Springer, ISBN 978-1-84800-014-8.
- Ren, W., Beard, R. W., and Atkins, E. M. (2007) Information consensus in multivehicle cooperative control: Collective group behavior through local interaction, *IEEE Control Systems Magazine*, 27(2), 71–82.
- Robert, J.-M. and Toussaint, G. T. (1990) Computational geometry and facility location, in *International Conference on Operations Research and Management Science*, pages 1–19, Manila, The Philippines.
- Roy, N. and Dudek, G. (2001) Collaborative exploration and rendezvous: Algorithms, performance bounds, and observations, *Autonomous Robots*, 11(2), 117–136.
- Sack, J. R. and Urrutia, J., (editors) (2000) *Handbook of Computational Geometry*, North-Holland, ISBN 0444825371.
- Sanfelice, R. G., Goebel, R., and Teel, A. R. (2007) Invariance principles for hybrid systems with connections to detectability and asymptotic stability, *IEEE Transactions on Automatic Control*, 52(12), 2282–2297.
- Santi, P. (2005) *Topology Control in Wireless Ad Hoc and Sensor Networks*, John Wiley, ISBN 0470094532.
- Santoro, N. (2001) Distributed computations by autonomous mobile robots, in *SOFSEM 2001: Conference on Current Trends in Theory and Practice of Informatics (Piestany, Slovak Republic)*, L. Pacholski and P. Ruzicka,

editors, volume 2234 of *Lecture Notes in Computer Science*, pages 110–115, Springer, ISBN 3-540-42912-3.

Sarlette, A. (2009) *Geometry and Symmetries in Coordination Control*, Ph.D. thesis, University of Liège, Belgium.

Sarlette, A. and Sepulchre, R. (2009) Consensus optimization on manifolds,
SIAM Journal on Control and Optimization, 48(1), 56–76.

- Savkin, A. (2004) Coordinated collective motion of groups of autonomous mobile robots: Analysis of Vicsek's model, *IEEE Transactions on Automatic Control*, 49(6), 981–982.
- Savla, K., Bullo, F., and Frazzoli, E. (2009a) Traveling Salesperson Problems for a double integrator, *IEEE Transactions on Automatic Control*, (Submitted Nov. 2006) to appear.
- Savla, K., Frazzoli, E., and Bullo, F. (2008) Traveling Salesperson Problems for the Dubins vehicle, *IEEE Transactions on Automatic Control*, 53(6), 1378–1391.
- Savla, K., Notarstefano, G., and Bullo, F. (2009b) Maintaining limited-range connectivity among second-order agents, *SIAM Journal on Control and Optimization*, 48(1), 187–205.
- Scardovi, L., Sarlette, A., and Sepulchre, R. (2007) Synchronization and balancing on the N -torus, *Systems & Control Letters*, 56(5), 335–341.
- Schultz, A. C. and Parker, L. E., (editors) (2002) *Multi-Robot Systems: From Swarms to Intelligent Automata*, Kluwer Academic Publishers, ISBN 1402006799, Proceedings from the 2002 NRL Workshop on Multi-Robot Systems.
- Schumacher, C., Chandler, P. R., Rasmussen, S. J., and Walker, D. (2003) Task allocation for wide area search munitions with variable path length, in *American Control Conference*, pages 3472–3477, Denver, CO.
- Schuresko, M. and Cortés, J. (2007) Safe graph rearrangements for distributed connectivity of robotic networks, in *IEEE Conference on Decision and Control*, pages 4602–4607, New Orleans, LA.
- Schwager, M., Bullo, F., Skelly, D., and Rus, D. (2008) A ladybug exploration strategy for distributed adaptive coverage control, in *IEEE International Conference on Robotics and Automation*, pages 2346–2353, Pasadena, CA.
- Schwager, M., Rus, D., and Slotine, J. J. (2009) Decentralized, adaptive coverage control for networked robots, *International Journal of Robotics Research*, to appear.
- Seeley, T. D. and Buhrman, S. C. (1999) Group decision-making in swarms of honey bees, *Behavioral Ecology and Sociobiology*, 45, 19–31.
- Seneta, E. (1981) *Non-negative Matrices and Markov Chains*, second edition, Springer, ISBN 0387297650.

Sepulchre, R., Paley, D. A., and Leonard, N. E. (2007) Stabilization of planar collective motion: All-to-all communication, *IEEE Transactions on Automatic Control*, 52(5), 811–824.

Sharma, V., Savchenko, M., Frazzoli, E., and Voulgaris, P. (2007) Transfer time complexity of conflict-free vehicle routing with no communications, *International Journal of Robotics Research*, 26(3), 255–272.

Sibson, R. (1981) A brief description of natural neighbour interpolation, in *Interpreting Multivariate Data*, V. Barnett, editor, pages 21–36, John Wiley, ISBN 0471280399.

Simmons, R., Apfelbaum, D., Fox, D., Goldman, R., Haigh, K., Musliner, D., Pelican, M., and Thrun, S. (2000) Coordinated deployment of multiple heterogeneous robots, in *IEEE/RSJ International Conference on Intelligent Robots & Systems*, pages 2254–2260, Takamatsu, Japan.

Sinclair, A. R. (1977) *The African Buffalo, A Study of Resource Limitation of Population*, The University of Chicago Press.

Sipser, M. (2005) *Introduction to the Theory of Computation*, second edition, Course Technology, ISBN 0534950973.

Skyum, S. (1991) A simple algorithm for computing the smallest enclosing circle, *Information Processing Letters*, 37(3), 121–125.

Smith, R. S. and Hadaegh, F. Y. (2007) Closed-loop dynamics of cooperative vehicle formations with parallel estimators and communication, *IEEE Transactions on Automatic Control*, 52(8), 1404–1414.

Smith, S. L., Broucke, M. E., and Francis, B. A. (2005) A hierarchical cyclic pursuit scheme for vehicle networks, *Automatica*, 41(6), 1045–1053.

—(2007) Curve shortening and the rendezvous problem for mobile autonomous robots, *IEEE Transactions on Automatic Control*, 52(6), 1154–1159.

Smith, S. L. and Bullo, F. (2009) Monotonic target assignment for robotic networks, *IEEE Transactions on Automatic Control*, 54(10), (Submitted June 2007) to appear.

Sontag, E. D. (1998) *Mathematical Control Theory: Deterministic Finite Dimensional Systems*, volume 6 of *TAM*, second edition, Springer, ISBN 0387984895.

—(2008) Input to state stability: Basic concepts and results, in *Nonlinear*

and Optimal Control Theory, P. Nistri and G. Stefani, editors, pages 163–220, Lecture Notes in Mathematics, Springer, ISBN 3540776443.

Spanos, D. P. and Murray, R. M. (2005) Motion planning with wireless network constraints, in *American Control Conference*, pages 87–92, Portland, OR.

- Spanos, D. P., Olfati-Saber, R., and Murray, R. M. (2005) Approximate distributed Kalman filtering in sensor networks with quantifiable performance, in *Symposium on Information Processing of Sensor Networks*, pages 133–139, Los Angeles, CA.
- Spong, M. W., Hutchinson, S., and Vidyasagar, M. (2006) *Robot Modeling and Control*, third edition, John Wiley, ISBN 0-471-64990-2.
- Stewart, K. J. and Harcourt, A. H. (1994) Gorillas vocalizations during rest periods – signals of impending departure, *Behaviour*, 130, 29–40.
- Strogatz, S. H. (2000) From Kuramoto to Crawford: Exploring the onset of synchronization in populations of coupled oscillators, *Physica D*, 143(1), 1–20.
- (2003) *SYNC: The emerging science of spontaneous order*, Hyperion, ISBN 0786868449.
- Sundaram, S. and Hadjicostis, C. N. (2008) Distributed function calculation and consensus using linear iterative strategies, *IEEE Journal on Selected Areas in Communications*, 26(4), 650–660.
- Susca, S. (2007) *Distributed Boundary Estimation and Monitoring*, Ph.D. thesis, University of California at Santa Barbara, available at <http://ccdc.mee.ucsb.edu>.
- Susca, S., Martínez, S., and Bullo, F. (2008) Monitoring environmental boundaries with a robotic sensor network, *IEEE Transactions on Control Systems Technology*, 16(2), 288–296.
- (2009) Gradient algorithms for polygonal approximation of convex contours, *Automatica*, 45(2), 510–516.
- Suzuki, I. and Yamashita, M. (1999) Distributed anonymous mobile robots: Formation of geometric patterns, *SIAM Journal on Computing*, 28(4), 1347–1363.
- Tabuada, P., Pappas, G. J., and Lima, P. (2005) Motion feasibility of multi-agent formations, *IEEE Transactions on Robotics*, 21(3), 387–392.
- Tahbaz-Salehi, A. and Jadbabaie, A. (2006) A one-

parameter family of distributed consensus algorithms with boundary: From shortest paths to mean hitting times, in *IEEE Conference on Decision and Control*, pages 4664–4669, San Diego, CA.

- (2007) Small world phenomenon, rapidly mixing markov chains, and average consensus algorithms, in *IEEE Conference on Decision and Control*, pages 276–281, New Orleans, LA.

— (2008) Consensus over random networks, *IEEE Transactions on Automatic Control*, 53(3), 791–795.

Tang, Z. and Özgüner, Ü. (2005) Motion planning for multi-target surveillance with mobile sensor agents, *IEEE Transactions on Robotics*, 21(5), 898–908.

Tanner, H. G., Jadbabaie, A., and Pappas, G. J. (2007) Flocking in fixed and switching networks, *IEEE Transactions on Automatic Control*, 52(5), 863–868.

Tanner, H. G., Pappas, G. J., and Kumar, V. (2004) Leader-to-formation stability, *IEEE Transactions on Robotics and Automation*, 20(3), 443–455.

Tel, G. (2001) *Introduction to Distributed Algorithms*, second edition, Cambridge University Press, ISBN 0521794838.

Toh, C.-K. (2001) *Ad Hoc Mobile Wireless Networks: Protocols and Systems*, Prentice Hall, ISBN 0130078174.

Triplett, B. I., Klein, D. J., and Morgansen, K. A. (2006) Discrete time Kuramoto models with delay, in *Network Embedded Sensing and Control. (Proceedings of NESC’05 Workshop)*, P.J. Antsaklis and P. Tabuada, editors, volume 331 of *Lecture Notes in Control and Information Sciences*, pages 9–24, Springer, ISBN 3540327940.

Tse, D. and Viswanath, P. (2005) *Fundamentals of Wireless Communication*, Cambridge University Press, ISBN 0521845270.

Tsitsiklis, J. N. (1984) *Problems in Decentralized Decision Making and Computation*, Ph.D. thesis, Massachusetts Institute of Technology, available at <http://web.mit.edu/jnt/www/Papers/PhD-84-jnt.pdf>.

Tsitsiklis, J. N., Bertsekas, D. P., and Athans, M. (1986) Distributed asynchronous deterministic and stochastic gradient optimization algorithms, *IEEE Transactions on Automatic Control*, 31(9), 803–812.

Tutuncu, R. H., Toh, K. C., and Todd, M. J. (2003) Solving semidefinite-quadratic-linear programs using SDPT3, *Mathematical Programming, Series B*, 95, 189–217.

van der Schaft, A. J. and Schumacher, H. (2000) *An Introduction to Hybrid Dynamical Systems*, volume 251 of *Lecture Notes in Control and Information Sciences*, Springer, ISBN 1852332336.

Vazirani, V. V. (2001) *Approximation Algorithms*, Springer, ISBN 3540653678.

297

- Vicsek, T., Czirók, A., Ben-Jacob, E., Cohen, I., and Shochet, O. (1995) Novel type of phase transition in a system of self-driven particles, *Physical Review Letters*, 75(6-7), 1226–1229.
- Wang, W. and Slotine, J.-J. E. (2006) A theoretical study of different leader roles in networks, *IEEE Transactions on Automatic Control*, 51(7), 1156–1161.
- Watton, A. and Kydon, D. W. (1969) Analytical aspects of the N -bug problem, *American Journal of Physics*, 37(2), 220–221.
- Whiteley, W. (1997) Rigidity and scene analysis, in *Handbook of Discrete and Computational Geometry*, J. E. Goodman and J. O'Rourke, editors, chapter 49, pages 893–916, CRC Press, ISBN 0849385245.
- Wiener, N. (1958) *Nonlinear Problems in Random Theory*, MIT Press.
- Winfree, A. T. (1980) *The Geometry of Biological Time*, Springer, ISBN 0387525289.
- Wolfowitz, J. (1963) Product of indecomposable, aperiodic, stochastic matrices, *Proceedings of American Mathematical Society*, 14(5), 733–737.
- Wu, C. W. (2006) Synchronization and convergence of linear dynamics in random directed networks, *IEEE Transactions on Automatic Control*, 51(7), 1207–1210.
- Xiao, L., Boyd, S., and Lall, S. (2005) A scheme for robust distributed sensor fusion based on average consensus, in *Symposium on Information Processing of Sensor Networks*, pages 63–70, Los Angeles, CA.
- Xue, F. and Kumar, P. R. (2004) The number of neighbors needed for connectivity of wireless networks, *Wireless Networks*, 10(2), 169–181.
- Yang, P., Freeman, R. A., and Lynch, K. M. (2008) Multi-agent coordination by decentralized estimation and control, *IEEE Transactions on Automatic Control*, 53(11), 2480–2496.
- Yu, C., Anderson, B. D. O., Dasgupta, S., and Fidan, B. (2009) Control of minimally persistent formations in the plane, *SIAM Journal on Control and Optimization*, 48(1), 206–233.
- Yu, J., LaValle, S. M., and Liberzon, D. (2008) Rendezvous without coordinates, in *IEEE Conference on Decision and Control*, pages 1803–1808, Cancún, México.

Zavlanos, M. M. and Pappas, G. J. (2005) Controlling connectivity of dynamic graphs, in *IEEE Conference on Decision and Control and European Control Conference*, pages 6388–6393, Seville, Spain.

- (2007a) Dynamic assignment in distributed motion planning with local information, in *American Control Conference*, pages 1173–1178, New York.
 - (2007b) Potential fields for maintaining connectivity of mobile networks, *IEEE Transactions on Robotics*, 23(4), 812–816.
- Zhang, F. and Leonard, N. E. (2005) Generating contour plots using multiple sensor platforms, in *IEEE Swarm Intelligence Symposium*, pages 309–316, Pasadena, CA.
- (2007) Coordinated patterns of unit speed particles on a closed curve, *Systems & Control Letters*, 56(6), 397–407.
- Zheng, Z., Spry, S. C., and Girard, A. R. (2008) Leaderless formation control using dynamic extension and sliding control, in *IFAC World Congress*, pages 16027–16032, Seoul, Korea.
- Zhong, M. and Cassandras, C. G. (2008) Distributed coverage control in sensor network environments with polygonal obstacles, in *IFAC World Congress*, pages 4162–4167, Seoul, Korea.
- Zhu, M. and Martínez, S. (2008a) Dynamic average consensus on synchronous communication networks, in *American Control Conference*, pages 4382–4387, Seattle, WA.
- (2008b) On the convergence time of distributed quantized averaging algorithms, in *IEEE Conference on Decision and Control*, pages 3971–3976, Cancún, México.