Concurrent and Distributed Computing in Java
This Page Intentionally Left Blank
Concurrent and Distributed Computing in Java
This Page Intentionally Left Blank
Concurrent and Distributed Computing in Java

Vijay K. Garg

University of Texas at Austin
To

my teachers and

my students
This Page Intentionally Left Blank
Contents

List of Figures xiii
Preface xix

1 Introduction 1
   1.1 Introduction ...................................... 1
   1.2 Distributed Systems versus Parallel Systems ............. 3
   1.3 Overview of the Book .................................. 4
   1.4 Characteristics of Parallel and Distributed Systems ..... 6
   1.5 Design Goals ......................................... 7
   1.6 Specification of Processes and Tasks ..................... 8
      1.6.1 Runnable Interface ................................ 11
      1.6.2 Join Construct in Java ........................... 11
      1.6.3 Thread Scheduling ................................ 13
   1.7 Problems ............................................. 13
   1.8 Bibliographic Remarks ................................ 15

2 Mutual Exclusion Problem 17
   2.1 Introduction .......................................... 17
   2.2 Peterson’s Algorithm .................................. 20
   2.3 Lamport’s Bakery Algorithm ............................ 24
   2.4 Hardware Solutions .................................... 27
      2.4.1 Disabling Interrupts ............................... 27
      2.4.2 Instructions with Higher Atomicity ................. 27
   2.5 Problems ............................................... 28
   2.6 Bibliographic Remarks ................................ 30

3 Synchronization Primitives 31
   3.1 Introduction .......................................... 31
   3.2 Semaphores ............................................ 31
3.2.1 The Producer-Consumer Problem ........................................... 33
3.2.2 The Reader-Writer Problem .............................................. 36
3.2.3 The Dining Philosopher Problem ......................................... 36
3.3 Monitors .............................................................................. 42
3.4 Other Examples ................................................................... 46
3.5 Dangers of Deadlocks ............................................................. 49
3.6 Problems .............................................................................. 50
3.7 Bibliographic Remarks .......................................................... 51

4 Consistency Conditions ............................................................... 53
4.1 Introduction ....................................................................... 53
4.2 System Model ...................................................................... 54
4.3 Sequential Consistency .......................................................... 55
4.4 Linearizability ...................................................................... 57
4.5 Other Consistency Conditions ............................................... 60
4.6 Problems ............................................................................. 62
4.7 Bibliographic Remarks .......................................................... 63

5 Wait-Free Synchronization .......................................................... 65
5.1 Introduction ....................................................................... 65
5.2 Safe, Regular, and Atomic Registers ....................................... 66
5.3 Regular SRSW Register ........................................................... 70
5.4 SRSW Multivalued Register .................................................... 71
5.5 MRSW Register .................................................................... 73
5.6 MRMW Register ................................................................... 74
5.7 Atomic Snapshots ................................................................ 76
5.8 Consensus ........................................................................... 78
5.9 Universal Constructions .......................................................... 84
5.10 Problems .......................................................................... 87
5.11 Bibliographic Remarks .......................................................... 87

6 Distributed Programming ............................................................. 89
6.1 Introduction ....................................................................... 89
6.2 inetAddress Class ................................................................. 89
6.3 Sockets based on UDP ............................................................ 90
   6.3.1 Datagram Sockets ............................................................ 90
   6.3.2 DatagramPacket Class ..................................................... 91
   6.3.3 Example Using Datagrams .............................................. 92
6.4 Sockets Based on TCP ............................................................. 94
   6.4.1 Server Sockets ............................................................... 96
<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.2.1 Chang–Roberts Algorithm</td>
<td>210</td>
</tr>
<tr>
<td>13.2.2 Hirschberg–Sinclair Algorithm</td>
<td>212</td>
</tr>
<tr>
<td>13.3 Election on General Graphs</td>
<td>213</td>
</tr>
<tr>
<td>13.3.1 Spanning Tree Construction</td>
<td>213</td>
</tr>
<tr>
<td>13.4 Application: Computing Global Functions</td>
<td>215</td>
</tr>
<tr>
<td>13.5 Problems</td>
<td>217</td>
</tr>
<tr>
<td>13.6 Bibliographic Remarks</td>
<td>219</td>
</tr>
</tbody>
</table>

14 Synchronizers | 221 |
| 14.1 Introduction | 221 |
| 14.2 A Simple Synchronizer | 223 |
| 14.2.1 Application: BFS Tree Construction | 225 |
| 14.3 Synchronizer α | 226 |
| 14.4 Synchronizer β | 228 |
| 14.5 Synchronizer γ | 230 |
| 14.6 Problems | 232 |
| 14.7 Bibliographic Remarks | 232 |

15 Agreement | 233 |
| 15.1 Introduction | 233 |
| 15.2 Consensus in Asynchronous Systems (Impossibility) | 234 |
| 15.3 Application: Terminating Reliable Broadcast | 238 |
| 15.4 Consensus in Synchronous Systems | 239 |
| 15.4.1 Consensus under Crash Failures | 240 |
| 15.4.2 Consensus under Byzantine Faults | 243 |
| 15.5 Knowledge and Common Knowledge | 244 |
| 15.6 Application: Two-General Problem | 248 |
| 15.7 Problems | 249 |
| 15.8 Bibliographic Remarks | 250 |

16 Transactions | 253 |
| 16.1 Introduction | 253 |
| 16.2 ACID Properties | 254 |
| 16.3 Concurrency Control | 255 |
| 16.4 Dealing with Failures | 256 |
| 16.5 Distributed Commit | 257 |
| 16.6 Problems | 261 |
| 16.7 Bibliographic Remarks | 262 |
CONTENTS

17 Recovery 263
17.1 Introduction ........................................... 263
17.2 Zigzag Relation ........................................... 265
17.3 Communication-Induced Checkpointing ......................... 267
17.4 Optimistic Message Logging: Main Ideas ......................... 268
  17.4.1 Model ............................................... 269
  17.4.2 Fault-Tolerant Vector Clock .......................... 270
  17.4.3 Version End Table ................................. 272
17.5 An Asynchronous Recovery Protocol .......................... 272
  17.5.1 Message Receive .................................... 274
  17.5.2 On Restart after a Failure ............................ 274
  17.5.3 On Receiving a Token .............................. 274
  17.5.4 On Rollback ....................................... 276
17.6 Problems ............................................... 277
17.7 Bibliographic Remarks .................................... 278

18 Self-Stabilization 279
18.1 Introduction ........................................... 279
18.2 Mutual Exclusion with K-State Machines ......................... 280
18.3 Self-Stabilizing Spanning Tree Construction ...................... 285
18.4 Problems ............................................... 286
18.5 Bibliographic Remarks .................................... 289

A. Various Utility Classes 291

Bibliography 297

Index 305
List of Figures

1.1 A parallel system ........................................ 2
1.2 A distributed system ..................................... 2
1.3 A process with four threads ............................ 9
1.4 HelloWorldThread.java .................................. 11
1.5 FooBar.java ................................................ 12
1.6 Fibonacci.java ............................................. 14

2.1 Interface for accessing the critical section ............ 18
2.2 A program to test mutual exclusion ........................ 19
2.3 An attempt that violates mutual exclusion ............... 20
2.4 An attempt that can deadlock ............................. 21
2.5 An attempt with strict alternation ........................ 21
2.6 Peterson’s algorithm for mutual exclusion ............... 22
2.7 Lamport’s bakery algorithm ............................... 25
2.8 TestAndSet hardware instruction .......................... 27
2.9 Mutual exclusion using TestAndSet ......................... 28
2.10 Semantics of swap operation .............................. 28
2.11 Dekker.java .................................................. 29

3.1 Binary semaphore ........................................... 32
3.2 Counting semaphore ....................................... 33
3.3 A shared buffer implemented with a circular array ...... 34
3.4 Bounded buffer using semaphores .......................... 35
3.5 Producer-consumer algorithm using semaphores .......... 37
3.6 Reader-writer algorithm using semaphores ............... 38
3.7 The dining philosopher problem ............................ 39
3.8 Dining Philosopher .......................................... 40
3.9 Resource Interface .......................................... 41
3.10 Dining philosopher using semaphores ....................... 41
3.11 A pictorial view of a Java monitor ........................ 44
LIST OF FIGURES

3.12 Bounded buffer monitor .................................... 45
3.13 Dining philosopher using monitors .......................... 47
3.14 Linked list .................................................. 48

4.1 Concurrent histories illustrating sequential consistency .... 56
4.2 Sequential consistency does not satisfy locality .............. 58
4.3 Summary of consistency conditions ........................... 62

5.1 Safe and unsafe read–write registers ........................ 67
5.2 Concurrent histories illustrating regularity .................... 68
5.3 Atomic and nonatomic registers ................................ 69
5.4 Construction of a regular boolean register .................... 71
5.5 Construction of a multivalued register ......................... 72
5.6 Construction of a multireader register ......................... 75
5.7 Construction of a multiwriter register ......................... 76
5.8 Lock-free atomic snapshot algorithm .......................... 77
5.9 Consensus Interface .......................................... 78
5.10 Impossibility of wait-free consensus with atomic read–write registers.............. 80
5.11 TestAndSet class .............................................. 81
5.12 Consensus using TestAndSet object ........................... 82
5.13 CompSwap object .............................................. 82
5.14 Consensus using CompSwap object ........................... 83
5.15 Load-Linked and Store-Conditional object .................... 84
5.16 Sequential queue .............................................. 85
5.17 Concurrent queue ............................................. 86

6.1 A datagram server .............................................. 93
6.2 A datagram client .............................................. 95
6.3 Simple name table ............................................. 97
6.4 Name server .................................................. 98
6.5 A client for name server ...................................... 99
6.6 Topology class ................................................. 100
6.7 Connector class ............................................... 102
6.8 Message class ................................................ 103
6.9 Linker class .................................................. 104
6.10 Remote interface ............................................. 105
6.11 A name service implementation ............................... 106
6.12 A RMI client program ....................................... 109

7.1 An example of topology of a distributed system ............... 113
7.2 A simple distributed program with two processes ............. 113
7.3 A run in the happened-before model .......................... 115
7.4 A logical clock algorithm ........................................ 117
7.5 A vector clock algorithm .......................................... 119
7.6 The VCLinker class that extends the Linker class .............. 120
7.7 A sample execution of the vector clock algorithm ............... 121
7.8 A direct-dependency clock algorithm ......................... 122
7.9 A sample execution of the direct-dependency clock algorithm ... 123
7.10 The matrix clock algorithm ...................................... 124

8.1 Testing a lock implementation .................................. 131
8.2 ListenableThread .................................................. 132
8.3 Process.java ...................................................... 133
8.4 A centralized mutual exclusion algorithm ...................... 134
8.5 Lamport's mutual exclusion algorithm ......................... 137
8.6 Ricart and Agrawala's algorithm .............................. 139
8.7 (a) Conflict graph; (b) an acyclic orientation with $P_2$ and $P_4$ as sources; (c) orientation after $P_2$ and $P_4$ finish eating .... 141
8.8 An algorithm for dining philosophers problem ................. 143
8.9 A token ring algorithm for the mutual exclusion problem ... 145

9.1 Consistent and inconsistent cuts ................................ 151
9.2 Classification of messages ....................................... 153
9.3 Chandy and Lamport's snapshot algorithm ..................... 155
9.4 Linker extended for use with SenderCamera ................... 158
9.5 A global snapshot algorithm based on sender recording ....... 159
9.6 Invocation of the global snapshot algorithm ................... 160

10.1 WCP (weak conjunctive predicate) detection algorithm—checker process ............................................... 167
10.2 Circulating token with vector clock ............................ 170
10.3 An application that runs circulating token with a sensor ...... 171
10.4 Monitor process algorithm at $P_i$ ................................ 172
10.5 Token-based WCP detection algorithm ......................... 174

11.1 A diffusing computation for the shortest path .................. 179
11.2 Interface for a termination detection algorithm .............. 179
11.3 Termination detection algorithm ................................ 183
11.4 A diffusing computation for the shortest path with termination ... 184
11.5 Termination detection by token traversal ....................... 186

12.1 A FIFO computation that is not causally ordered .......... 191
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.2</td>
<td>An algorithm for causal ordering of messages at $P_1$</td>
<td>193</td>
</tr>
<tr>
<td>12.3</td>
<td>Structure of a causal message</td>
<td>194</td>
</tr>
<tr>
<td>12.4</td>
<td>CausalLinker for causal ordering of messages</td>
<td>195</td>
</tr>
<tr>
<td>12.5</td>
<td>A chat program</td>
<td>197</td>
</tr>
<tr>
<td>12.6</td>
<td>A computation that is synchronously ordered</td>
<td>198</td>
</tr>
<tr>
<td>12.7</td>
<td>A computation that is not synchronously ordered</td>
<td>198</td>
</tr>
<tr>
<td>12.8</td>
<td>The algorithm at $P_1$ for synchronous ordering of messages</td>
<td>201</td>
</tr>
<tr>
<td>12.9</td>
<td>The algorithm for synchronous ordering of messages</td>
<td>202</td>
</tr>
<tr>
<td>13.1</td>
<td>The leader election algorithm</td>
<td>211</td>
</tr>
<tr>
<td>13.2</td>
<td>Configurations for the worst case (a) and the best case (b)</td>
<td>212</td>
</tr>
<tr>
<td>13.3</td>
<td>A spanning tree construction algorithm</td>
<td>214</td>
</tr>
<tr>
<td>13.4</td>
<td>A convergecast algorithm</td>
<td>216</td>
</tr>
<tr>
<td>13.5</td>
<td>A broadcast algorithm</td>
<td>216</td>
</tr>
<tr>
<td>13.6</td>
<td>Algorithm for computing a global function</td>
<td>218</td>
</tr>
<tr>
<td>13.7</td>
<td>Computing the global sum</td>
<td>219</td>
</tr>
<tr>
<td>14.1</td>
<td>Algorithm for the simple synchronizer at $P_2$</td>
<td>223</td>
</tr>
<tr>
<td>14.2</td>
<td>Implementation of the simple synchronizer</td>
<td>224</td>
</tr>
<tr>
<td>14.3</td>
<td>An algorithm that generates a tree on an asynchronous network</td>
<td>226</td>
</tr>
<tr>
<td>14.4</td>
<td>BFS tree algorithm using a synchronizer</td>
<td>227</td>
</tr>
<tr>
<td>14.5</td>
<td>Alpha synchronizer</td>
<td>229</td>
</tr>
<tr>
<td>15.1</td>
<td>(a) Commutativity of disjoint events; (b) asynchrony of messages</td>
<td>234</td>
</tr>
<tr>
<td>15.2</td>
<td>(a) Case 1: $\text{proc}(e) \neq \text{proc}(f)$; (b) case 2: $\text{proc}(e) = \text{proc}(f)$</td>
<td>237</td>
</tr>
<tr>
<td>15.3</td>
<td>Algorithm at $P_1$ for consensus under crash failures</td>
<td>241</td>
</tr>
<tr>
<td>15.4</td>
<td>Consensus in a synchronous environment</td>
<td>242</td>
</tr>
<tr>
<td>15.5</td>
<td>Consensus tester</td>
<td>243</td>
</tr>
<tr>
<td>15.6</td>
<td>An algorithm for Byzantine General Agreement</td>
<td>245</td>
</tr>
<tr>
<td>16.1</td>
<td>Algorithm for the coordinator of the two-phase commit protocol</td>
<td>259</td>
</tr>
<tr>
<td>16.2</td>
<td>Algorithm for the participants in the two-phase commit protocol</td>
<td>260</td>
</tr>
<tr>
<td>17.1</td>
<td>An example of the domino effect</td>
<td>264</td>
</tr>
<tr>
<td>17.2</td>
<td>Examples of zigzag paths</td>
<td>266</td>
</tr>
<tr>
<td>17.3</td>
<td>A distributed computation</td>
<td>271</td>
</tr>
<tr>
<td>17.4</td>
<td>Formal description of the fault-tolerant vector clock</td>
<td>273</td>
</tr>
<tr>
<td>17.5</td>
<td>Formal description of the version end-table mechanism</td>
<td>273</td>
</tr>
<tr>
<td>17.6</td>
<td>An optimistic protocol for asynchronous recovery</td>
<td>275</td>
</tr>
<tr>
<td>18.1</td>
<td>$K$-state self-stabilizing algorithm</td>
<td>280</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

18.2 A move by the bottom machine in the K-state algorithm .................. 280
18.3 A move by a normal machine in the K-state algorithm .................. 281
18.4 Self-stabilizing algorithm for mutual exclusion in a ring for the bottom 
    machine ........................................ 283
18.5 Self-stabilizing algorithm for mutual exclusion in a ring for a normal 
    machine ........................................ 284
18.6 Self-stabilizing algorithm for (BFS) spanning tree ....................... 285
18.7 Self-stabilizing spanning tree algorithm for the root .................... 286
18.8 Self-stabilizing spanning tree algorithm for nonroot nodes ............ 287
18.9 A Java program for spanning tree ..................................... 288

A.1 Util.java ........................................................................ 292
A.2 Symbols.java .............................................................. 293
A.3 Matrix.java ................................................................... 293
A.4 MsgList.java .............................................................. 294
A.5 IntLinkedList.java ......................................................... 294
A.6 PortAddr.java .............................................................. 295
Preface

This book is designed for a senior undergraduate-level course or an introductory graduate-level course on concurrent and distributed computing. This book grew out of my dissatisfaction with books on distributed systems (including books authored by me) that included pseudocode for distributed algorithms. There were two problems with pseudocode. First, pseudocode had many assumptions hidden in it making it more succinct but only at the expense of precision. Second, translating pseudocode into actual code requires effort and time, resulting in students never actually running the algorithm. Seeing the code run lends an extra level of confidence in one’s understanding of the algorithms.

It must be emphasized that all of the Java code provided in this book is for educational purposes only. I have deliberately avoided error checking and other software engineering principles to keep the size of the code small. In the majority of cases, this led to Java code, that kept the concepts of the algorithm transparent. Several examples and exercise problems are included in each chapter to facilitate classroom teaching. I have made an effort to include some programming exercises with each chapter.

I would like to thank the following people for working with me on various projects discussed in this book: Craig Chase (weak predicates), Om Damani (message logging), Eddy Fromentin (predicate detection), Joydeep Ghosh (global computation), Richard Kilgore (channel predicates), Roger Mitchell (channel predicates), Neeraj Mittal (predicate detection and control, slicing, self-stabilization, distributed shared memory), Venkat Murty (synchronous ordering), Michel Raynal (control flow properties, distributed shared memory), Alper Sen (slicing), Chakarat Skawratonand (vector clocks), Ashis Tarafdar (message logging, predicate control), Alexander Tomlinson (global time, mutual exclusion, relational predicates, control flow properties) and Brian Waldecker (weak and strong predicates). Anurag Agarwal, Arindam Chakraborty, Selma Ikiz, Neeraj Mittal, Sujatha Kashyap, Vinit Ogale, and Alper Sen reviewed parts of the book. I owe special thanks to Vinit Ogale for also helping me with figures.

I thank the Department of Electrical and Computer Engineering at The Uni-
versity of Texas at Austin, where I was given the opportunity to develop and teach courses on concurrent and distributed systems. Students in these courses gave me very useful feedback.

I was supported in part by many grants from the National Science Foundation over the last 14 years. Many of the results reported in this book would not have been discovered by me and my research group without that support. I also thank John Wiley & Sons, Inc. for supporting the project.

Finally, I thank my parents, wife and children. Without their love and support, this book would not have been even conceived.

There are many concurrent and distributed programs in this book. Although I have tried to ensure that there are no "bugs" in these programs, some are, no doubt, still lurking in the code. I would be grateful if any bug that is discovered is reported to me. The list of known errors and the supplementary material for the book will be maintained on my homepage:

http://www.ece.utexas.edu/~garg

Included in the Website is a program that allows animation of most of the algorithms in the book. It also includes all the source code given in the book. The reader can access the source code with the user name as guest and the password as utexas.

Vijay K. Garg
Austin, Texas
Chapter 1

Introduction

1.1 Introduction

Parallel and distributed computing systems are now widely available. A parallel system consists of multiple processors that communicate with each other using shared memory. As the number of transistors on a chip increases, multiprocessor chips will become fairly common. With enough parallelism available in applications, such systems will easily beat sequential systems in performance. Figure 1.1 shows a parallel system with multiple processors. These processors communicate with each other using the shared memory. Each processor may also have local memory that is not shared with other processors.

We define distributed systems as those computer systems that contain multiple processors connected by a communication network. In these systems processors communicate with each other using messages that are sent over the network. Such systems are increasingly available because of decrease in prices of computer processors and the high-bandwidth links to connect them. Figure 1.2 shows a distributed system. The communication network in the figure could be a local area network such as an Ethernet, or a wide area network such as the Internet.

Programming parallel and distributed systems requires a different set of tools and techniques than that required by the traditional sequential software. The focus of this book is on these techniques.
CHAPTER 1. INTRODUCTION

Figure 1.1: A parallel system

Figure 1.2: A distributed system
1.2 Distributed Systems versus Parallel Systems

In this book, we make a distinction between distributed systems and parallel systems. This distinction is only at a logical level. Given a physical system in which processors have shared memory, it is easy to simulate messages. Conversely, given a physical system in which processors are connected by a network, it is possible to simulate shared memory. Thus a parallel hardware system may run distributed software and vice versa.

This distinction raises two important questions. Should we build parallel hardware or distributed hardware? Should we write applications assuming shared memory or message passing? At the hardware level, we would expect the prevalent model to be multiprocessor workstations connected by a network. Thus the system is both parallel and distributed. Why would the system not be completely parallel? There are many reasons.

- **Scalability**: Distributed systems are inherently more scalable than parallel systems. In parallel systems shared memory becomes a bottleneck when the number of processors is increased.

- **Modularity and heterogeneity**: A distributed system is more flexible because a single processor can be added or deleted easily. Furthermore, this processor can be of a type completely different from that of the existing processors.

- **Data sharing**: Distributed systems provide data sharing as in distributed databases. Thus multiple organizations can share their data with each other.

- **Resource sharing**: Distributed systems provide resource sharing. For example, an expensive special-purpose processor can be shared by multiple organizations.

- **Geographic structure**: The geographic structure of an application may be inherently distributed. The low communication bandwidth may force local processing. This is especially true for wireless networks.

- **Reliability**: Distributed systems are more reliable than parallel systems because the failure of a single computer does not affect the availability of others.

- **Low cost**: Availability of high-bandwidth networks and inexpensive workstations also favors distributed computing for economic reasons.

Why would the system not be a purely distributed one? The reasons for keeping a parallel system at each node of a network are mainly technological in nature. With the current technology it is generally faster to update a shared memory location than
to send a message to another processor. This is especially true when the new value of the variable must be communicated to multiple processors. Consequently, it is more efficient to get fine-grain parallelism from a parallel system than from a distributed system.

So far our discussion has been at the hardware level. As mentioned earlier, the interface provided to the programmer can actually be independent of the underlying hardware. So which model would then be used by the programmer? At the programming level, we expect that programs will be written using multithreaded distributed objects. In this model, an application consists of multiple heavyweight processes that communicate using messages (or remote method invocations). Each heavyweight process consists of multiple lightweight processes called threads. Threads communicate through the shared memory. This software model mirrors the hardware that is (expected to be) widely available. By assuming that there is at most one thread per process (or by ignoring the parallelism within one process), we get the usual model of a distributed system. By restricting our attention to a single heavyweight process, we get the usual model of a parallel system. We expect the system to have aspects of distributed objects. The main reason is the logical simplicity of the distributed object model. A distributed program is more object-oriented because data in a remote object can be accessed only through an explicit message (or a remote procedure call). The object orientation promotes reusability as well as design simplicity. Furthermore, these objects would be multithreaded because threads are useful for implementing efficient objects. For many applications such as servers, it is useful to have a large shared data structure. It is a programming burden and inefficient to split the data structure across multiple heavyweight processes.

1.3 Overview of the Book

This book is intended for a one-semester advanced undergraduate or introductory graduate course on concurrent and distributed systems. It can also be used as a supplementary book in a course on operating systems or distributed operating systems. For an undergraduate course, the instructor may skip the chapters on consistency conditions, wait-free synchronization, synchronizers, recovery, and self-stabilization without any loss of continuity.

Chapter 1 provides the motivation for parallel and distributed systems. It compares advantages of distributed systems with those of parallel systems. It gives the defining characteristics of parallel and distributed systems and the fundamental difficulties in designing algorithms for such systems. It also introduces basic constructs of starting threads in Java.

Chapters 2–5 deal with multithreaded programming. Chapter 2 discusses the
mutual exclusion problem in shared memory systems. This provides motivation to students for various synchronization primitives discussed in Chapter 3. Chapter 3 exposes students to multithreaded programming. For a graduate course, Chapters 2 and 3 can be assigned for self-study. Chapter 4 describes various consistency conditions on concurrent executions that a system can provide to the programmers. Chapter 5 discusses a method of synchronization which does not use locks. Chapters 4 and 5 may be skipped in an undergraduate course.

Chapter 6 discusses distributed programming based on sockets as well as remote method invocations. It also provides a layer for distributed programming used by the programs in later chapters. This chapter is a prerequisite to understanding programs described in later chapters.

Chapter 7 provides the fundamental issues in distributed programming. It discusses models of a distributed system and a distributed computation. It describes the *interleaving model* that totally orders all the events in the system, and the *happened before model* that totally orders all the events on a single process. It also discusses mechanisms called *clocks* used to timestamp events in a distributed computation such that order information between events can be determined with these clocks. This chapter is fundamental to distributed systems and should be read before all later chapters.

Chapter 8 discusses one of the most studied problems in distributed systems—mutual exclusion. This chapter provides the interface *Lock* and discusses various algorithms to implement this interface. *Lock* is used for coordinating resources in distributed systems.

Chapter 9 discusses the abstraction called *Camera* that can be used to compute a consistent snapshot of a distributed system. We describe Chandy and Lamport's algorithm in which the receiver is responsible for recording the state of a channel as well as a variant of that algorithm in which the sender records the state of the channel. These algorithms can also be used for detecting stable global properties—properties that remain true once they become true.

Chapters 10 and 11 discuss the abstraction called *Sensor* that can be used to evaluate global properties in a distributed system. Chapter 10 describes algorithms for detecting conjunctive predicates in which the global predicate is simply a conjunction of local predicates. Chapter 11 describe algorithms for termination and deadlock detection. Although termination and deadlock can be detected using techniques described in Chapters 9 and 10, we devote a separate chapter for termination and deadlock detection because these algorithms are more efficient than those used to detect general global properties. They also illustrate techniques in designing distributed algorithms.

Chapter 12 describe methods to provide messaging layer with stronger properties than provided by the Transmission Control Protocol (TCP). We discuss the causal
ordering of messages, the synchronous and the total ordering of messages.

Chapter 13 discusses two abstractions in a distributed system—*Election* and *GlobalFunction*. We discuss election in ring-based systems as well as in general graphs. Once a leader is elected, we show that a global function can be computed easily via a convergecast and a broadcast.

Chapter 14 discusses synchronizers, a method to abstract out asynchrony in the system. A synchronizer allows a synchronous algorithm to be simulated on top of an asynchronous system. We apply synchronizers to compute the breadth-first search (BFS) tree in an asynchronous network.

Chapters 1–14 assume that there are no faults in the system. The rest of the book deals with techniques for handling various kinds of faults.

Chapter 15 analyze the possibility (or impossibility) of solving problems in the presence of various types of faults. It includes the fundamental impossibility result of Fischer, Lynch, and Paterson that shows that consensus is impossible to solve in the presence of even one unannounced failure in an asynchronous system. It also shows that the consensus problem can be solved in a synchronous environment under crash and Byzantine faults. It also discusses the ability to solve problems in the absence of reliable communication. The two-generals problem shows that agreement on a bit (gaining common knowledge) is impossible in a distributed system.

Chapter 16 describes the notion of a transaction and various algorithms used in implementing transactions.

Chapter 17 discusses methods of recovering from failures. It includes both checkpointing and message-logging techniques.

Finally, Chapter 18 discusses self-stabilizing systems. We discuss solutions of the mutual exclusion problem when the state of any of the processors may change arbitrarily because of a fault. We show that it is possible to design algorithms that guarantee that the system converges to a legal state in a finite number of moves irrespective of the system execution. We also discuss self-stabilizing algorithms for maintaining a spanning tree in a network.

There are numerous starred and unstarred problems at the end of each chapter. A student is expected to solve unstarred problems with little effort. The starred problems may require the student to spend more effort and are appropriate only for graduate courses.

### 1.4 Characteristics of Parallel and Distributed Systems

Recall that we distinguish between parallel and distributed systems on the basis of shared memory. A distributed system is characterized by absence of shared memory. Therefore, in a distributed system it is impossible for any one processor to know
the global state of the system. As a result, it is difficult to observe any global property of the system. We will later see how efficient algorithms can be developed for evaluating a suitably restricted set of global properties.

A parallel or a distributed system may be tightly coupled or loosely coupled depending on whether multiple processors work in a lock step manner. The absence of a shared clock results in a loosely coupled system. In a geographically distributed system, it is impossible to synchronize the clocks of different processors precisely because of uncertainty in communication delays between them. As a result, it is rare to use physical clocks for synchronization in distributed systems. In this book we will see how the concept of causality is used instead of time to tackle this problem. In a parallel system, although a shared clock can be simulated, designing a system based on a tightly coupled architecture is rarely a good idea, due to loss of performance because of synchronization. In this book, we will assume that systems are loosely coupled.

Distributed systems can further be classified into synchronous and asynchronous systems. A distributed system is asynchronous if there is no upper bound on the message communication time. Assuming asynchrony leads to most general solutions to various problems. We will see many examples in this book. However, things get difficult in asynchronous systems when processors or links can fail. In an asynchronous distributed system it is impossible to distinguish between a slow processor and a failed processor. This leads to difficulties in developing algorithms for consensus, election, and other important problems in distributed computing. We will describe these difficulties and also show algorithms that work under faults in synchronous systems.

1.5 Design Goals

The experience in large parallel and distributed software systems has shown that their design should take the following concepts into consideration [TvS02]:

- **Fault tolerance**: The software system should mask the failure of one or more components in the system, including processors, memory, and network links. This generally requires redundancy, which may be expensive depending on the degree of fault tolerance. Therefore, cost–benefit analysis is required to determine an appropriate level of fault tolerance.

- **Transparency**: The system should be as user-friendly as possible. This requires that the user not have to deal with unnecessary details. For example, in a heterogeneous distributed system the differences in the internal representation of data (such as the little endian format versus the big endian format for
integers) should be hidden from the user, a concept called access transparency. Similarly, the use of a resource by a user should not require the user to know where it is located (location transparency), whether it is replicated (replication transparency), whether it is shared (concurrency transparency), or whether it is in volatile memory or hard disk (persistence transparency).

- **Flexibility**: The system should be able to interact with a large number of other systems and services. This requires that the system adhere to a fixed set of rules for syntax and semantics, preferably a standard, for interaction. This is often facilitated by specification of services provided by the system through an interface definition language. Another form of flexibility can be given to the user by a separation between policy and mechanism. For example, in the context of Web caching, the mechanism refers to the implementation for storing the Web pages locally. The policy refers to the high-level decisions such as size of the cache, which pages are to be cached, and how long those pages should remain in the cache. Such questions may be answered better by the user and therefore it is better for users to build their own caching policy on top of the caching mechanism provided. By designing the system as one monolithic component, we lose the flexibility of using different policies with different users.

- **Scalability**: If the system is not designed to be scalable, then it may have unsatisfactory performance when the number of users or the resources increase. For example, a distributed system with a single server may become overloaded when the number of clients requesting the service from the server increases. Generally, the system is either completely decentralized using distributed algorithms or partially decentralized using a hierarchy of servers.

### 1.6 Specification of Processes and Tasks

In this book we cover the programming concepts for shared memory-based languages and distributed languages. It should be noted that the issues of concurrency arise even on a single CPU computer where a system may be organized as a collection of cooperating processes. In fact, the issues of synchronization and deadlock have roots in the development of early operating systems. For this reason, we will refer to constructs described in this section as concurrent programming.

Before we embark on concurrent programming constructs, it is necessary to understand the distinction between a *program* and a *process*. A computer program is simply a set of instructions in a high-level or a machine-level language. It is only when we execute a program that we get one or more *processes*. When the program is