Chapter 1

Introduction

Distributed System: You know you have one when the crash of a computer you've never heard of stops you from getting any work done.
— Leslie Lamport

1.1 Introduction

This book is on distributed systems. 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 decreases prices of computer processors and the availability of high-bandwidth links to connect them. Despite the availability of hardware for distributed systems, there are few software applications that exploit the hardware. One reason is that distributed software 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.

1.2 Distributed Systems Versus Parallel Systems

In this book, we make a distinction between distributed systems and parallel systems. A parallel system consists of multiple processors that communicate with each other using shared memory. 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 not? At the hardware level, we would expect that the prevalent model would 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 completely different type than 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.

- *Geographical structure*: The geographical 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 purely a distributed one? The reasons for keeping a parallel system at each node are mainly of a technological nature. With the current technology it is generally faster to update a shared memory location than to send a message to some other 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 heavy-weight processes that communicate using messages (or remote method invocations). Each heavy-weight process consists of multiple light-weight 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 heavy-weight process, we get the usual model of a parallel system. Why would the system 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 only be accessed through an explicit message (or a remote procedure call). The object orientation promotes reusability as well as design simplicity. Conversely, threads are also useful to provide efficient objects. For many applications such as servers, it is useful to have a large shared data structure. It is a programming burden to split the data structure across multiple heavy-weight processes.

In summary, we will see aspects of both parallel processing and distributed processing in hardware as well as software. This book is mainly about techniques and tools for distributed software.

1.3 Characteristics of Distributed Systems

We take the following characteristics as the defining ones for distributed systems.

- *Absence of a shared clock*: In a 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.

- *Absence of shared memory*: 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. In this book we will see how efficient algorithms can be developed for evaluating a suitably restricted set of global properties.

- *Absence of accurate failure detection*: In an asynchronous distributed system (a distributed system is asynchronous if there is no upper bound on the message communication time), it is impossible to distinguish between a slow processor and a failed processor. This leads to many difficulties in developing algorithms for consensus, election, etc. In this book we will see how failure detectors can be built to alleviate some of these problems.

1.4 Scope of the Book

This book discusses fundamental concepts in distributed computing systems such as time, state, simultaneity, order, knowledge, failure, and agreement in distributed systems. The emphasis of the book is on developing general mechanisms that can be applied to variety of problems. Examples are clocks, locks, cameras, sensors, controllers, slicers, and synchronizers. The topics have been carefully chosen so that they are fundamental yet useful in practical contexts. The emphasis is on positive results (algorithms) rather than on negative results (lower bounds and impossibility).

The book is based on an *asynchronous* model of distributed computing. Thus it does not deal with shared memory models or network computing with bounded delays on messages and bounded delay between consecutive steps of a processor. An algorithm developed for an asynchronous model of distributed computing will work correctly on all communication networks. In contrast, a *synchronous* algorithm

may not work correctly when message delays exceed the upper bound or when processors do not provide hard real-time guarantees.

1.5 Overview of the Book

This book is intended for a one-semester advanced undergraduate or an introductory graduate course on distributed systems. Each chapter can be covered in one 75-minute lecture. There are exactly thirty chapters in the book, making it sufficient for a fifteen-week semester.

There is very little dependence across chapters so that the instructor can pick and choose chapters that he or she wants to cover in the course.

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

The rest of the chapters are organized as shown in Figure 1.1.

Chapter 2 discusses models of a distributed computation. It describes three models. The first model, called the *interleaving model*, totally orders all the events in the system; the second model, called the *happened before model*, totally orders all the events on a single process; and the third model, called the *potential causality mode*, assumes only a partial order on events even within a single process. This chapter is fundamental to the entire book and should be read before all other chapters.

Chapters 3–22 assume that there are no faults in the system whereas Chapters 23–30 describe solutions (or impossibility thereof) for various problems under various kinds of faults.

Chapters 3–5 discuss mechanisms called clocks used to timestamp events in a distributed computation such that order information between events can be determined with these clocks. Chapter 3 discusses logical and vector clocks. Chapter 4 gives a formal proof of correctness of a vector clock algorithm. This chapter may be skipped without any loss of continuity. Chapter 5 describes clocks of different dimensions, such as matrix clocks.

Chapters 6–9 discuss problems that arise in coordinating resources. Chapters 6 and 7 discuss one of the most studied problems in distributed systems—mutual exclusion. Chapter 6 presents mutual exclusion algorithms based on timestamping, whereas Chapter 7 presents token-based mutual exclusion algorithms. The goal of Chapter 6 is

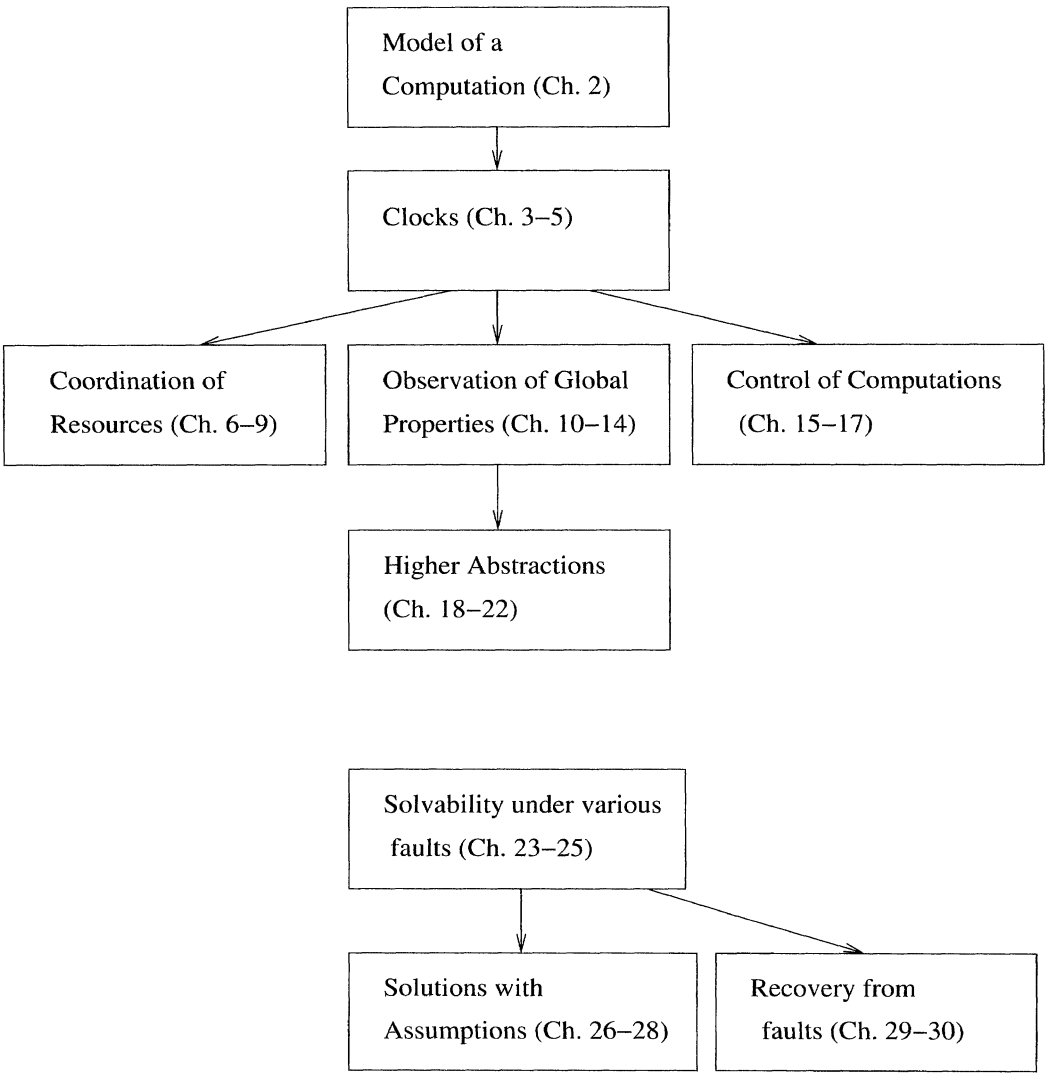

Figure 1.1: Organization of the book

to give the reader a flavor of methods in specification and verification of properties of distributed algorithms. In particular, Chapter 6 emphasizes a method for specification and verification that is based on the structural induction on the partially ordered set generated by a distributed program. The goal of Chapter 7 is to show how a centralized algorithm can be decentralized based on the notion of tokens. It also presents quorum-based algorithms for mutual exclusion. Chapter 8 discusses a generalization of the mutual exclusion problem called the drinking philosophers problem. Chapter 9 describes algorithms for leader election. Leader election is useful for resource coordination problems to implement a centralized coordinator scheme.

Chapters 10–14 discuss observation of global properties. In Chapter 10, we describe Chandy and Lamport's algorithm to compute the global state of a distributed system. Our proof of correctness of Chandy and Lamport's algorithm is based on the structure of the poset of the distributed computation. This proof is different from the original proof based on execution sequence, as provided by Chandy and Lamport. This algorithm can be used for detecting stable global properties—properties that remain true once they become true. Chapter 11 discusses the main techniques for detecting unstable predicates. The general problem of detecting unstable predicates is shown to be NP-complete; however, if the given predicate satisfies certain properties, then it can be efficiently detected. Chapter 12 and 13 discuss two important classes of unstable predicates—conjunctive predicates and channel predicates. Chapter 14 presents a variant of Dijkstra and Scholten's algorithm for termination detection. Chapters 13 and 14 can be skipped without any loss of continuity.

Chapters 15–17 discuss issues in controlling distributed computations. Chapter 15 discusses a method of off-line control in which appropriate control messages are inserted in a computation to guarantee invariance of a given predicate such as nonviolation of mutual exclusion. Chapter 16 describes methods to provide the causal ordering of messages, and Chapter 17 describes the synchronous and total ordering of messages.

Chapters 18–22 describe various higher-level abstractions and tools that can be used to build distributed applications. Chapter 18 and 19 discuss methods to compute a global function in a network either just once or repeatedly. By using various methods to gather and disseminate data, these abstractions hide the fact that data is distributed. The intent is to develop a framework that can be applied to a variety

of problems in a distributed system. For example, the techniques presented in this chapter can be used to compute centers of a network, to compute fixed points of equations, and to solve problems using branch-and-bound techniques. Chapter 20 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. Chapter 21 discusses slicing, a tool to combat computational complexity of analysis of distributed computations. Chapter 22 describes methods to build distributed shared memory, thereby allowing parallel algorithms to be simulated on top of distributed systems.

Chapters 23–25 analyze possibility (or impossibility) of solving problems in the presence of various types of faults. Chapter 23 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. Chapter 24 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 25 discusses the ability to solve problems when processors may crash. It includes the fundamental impossibility result of Fischer, Lynch and Patterson that shows that consensus is impossible to solve in the presence of even one unannounced failure.

Chapters 26–28 solve problems in the presence of faults either by making stronger assumptions on the environment or by weakening the requirements of the problem. Chapter 26 shows that the consensus problem can be solved in a synchronous environment under crash and Byzantine faults. Chapter 27 shows that it can be solved assuming the presence of failure detectors. Chapter 28 shows some problems that can be solved in an asynchronous environment.

Chapters 29 and 30 discuss methods of recovering from failures when failure detection is perfect.

Finally, the Appendix gives a concise introduction to the concepts in partially ordered sets and lattices.

There are a large number of 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 either spend more effort or read the cited paper.

1.6 Notation

We use the following notation for quantified expressions:

$$(op \text{ free-var-list} : \text{range-of-free-vars} : \text{expression})$$

where *op* is a universal or an existential quantifier, free-var-list is the list of variables over which the quantification is made, and the range-of-free-vars is the range of the variables. For example, $(\forall i : 0 \leq i \leq 10 : i^2 \leq 100)$ means that for all i such that $0 \leq i \leq 10$, $i^2 \leq 100$ holds. If the range of the variables is clear from the context, then we simply use:

$$(op \text{ free-var-list} : \text{expression})$$

For example, if it is clear that i and j are integers then we may write

$$\forall i : (\exists j : j > i)$$

We use a calculational style of proofs for many of our theorems. For example, a proof that $[A \equiv C]$ is rendered in our format as

$$\begin{array}{ll} & A \\ \equiv & \quad \{ \text{ hint why } [A \equiv B] \ \} \\ & B \\ \equiv & \quad \{ \text{ hint why } [B \equiv C] \ \} \\ & C. \end{array}$$

We use implication ($\Rightarrow$) instead of equivalence when proving $A \Rightarrow C$.

A predicate with free variables is assumed to be universally quantified for all possible values of free variables. We use the usual convention for binding powers of operators. In order of increasing binding powers, the operators are:

$\equiv$
$\Rightarrow$
$\vee, \wedge$
$\neg$
$=, \neq, <, \leq$, and other relational operators over integers and sets, arithmetic operators, and
function applications.

Operators that appear on the same line have the same binding power, and we use parentheses to show the order of application. We sometimes

omit parentheses in expressions when they use operators from different lines. Thus

$$s \rightarrow t \Rightarrow C(s) < C(t)$$

is equivalent to

$$\forall s, t : ((s \rightarrow t) \Rightarrow (C(s) < C(t))).$$

1.7 Problems

1.1. Give advantages and disadvantages of a parallel programming model over a distributed system (message based) model.

1.2. Compare the mechanisms for *synchronization* and *communication* in parallel programming languages and distributed programming languages.

1.8 Bibliographic Remarks

Many books are available on distributed systems. The reader is referred to books by Attiya and Welch [AW98], Barbosa [Bar96], Chandy and Misra [CM89], Garg [Gar96], Lynch [Lyn96], Raynal [Ray88], and Tel [Tel94] for the range of topics in distributed algorithms. Couloris, Dollimore, and Kindberg [CDK94] and Chow and Johnson [CJ97] cover some practical aspects of distributed systems, such as distributed file systems, that are not covered in this book. Goscinski [Gos91] and Singhal and Shivaratri [SS94] cover concepts in distributed operating systems. Some edited books are also available. The book edited by Yang and Marsland [YM94] includes many papers that deal with global time and state in distributed systems. The book edited by Mullender [Mul94] covers many other topics such as protection, fault tolerance, and real-time communications. The proof format adopted for many theorems in this book is taken from Dijkstra and Scholten [DS90].