1

Introduction: Distributed Systems

This chapter gives reasons for the study of distributed algorithms by briefly introducing the types of hardware and software systems for which distributed algorithms have been developed. By a distributed system we mean all computer applications where several computers or processors cooperate in some way. This definition includes wide-area computer communication networks, but also local-area networks, multiprocessor computers in which each processor has its own control unit, and systems of cooperating processes.

The different types of distributed system and the reasons why distributed systems are used are discussed in Section 1.1. Some examples of existing systems will be given. The main topic of this book, however, is not what these systems look like, or how they are used, but how they can be made to work. And even that topic will be further specialized towards the treatment of the *algorithms* used in the systems.

Of course, the entire structure and operation of a distributed system is not fully understood by a study of its algorithms alone. To understand such a system fully one must also study the complete architecture of its hardware and software, that is, the partition of the entire functionality into modules. Also, there are many important questions related to properties of the programming languages used to build the software of distributed systems. These subjects will be discussed in Section 1.2.

It is the case, however, that there are already in existence excellent books about distributed systems, which concentrate on the architectural and language aspects; see, e.g., Tanenbaum [Tan96], Sloman and Kramer [SK87], Bal [Bal90], Coulouris and Dollimore [CD88] or Goscinski [Gos91]. As already mentioned, the present text concentrates on algorithms for distributed systems. Section 1.3 explains why the design of distributed algorithms dif-

fers from the design of centralized algorithms, sketches the research field of distributed algorithms, and outlines the remainder of the book.

1.1 What is a Distributed System?

In this chapter we shall use the term "distributed system" to mean an interconnected collection of autonomous computers, processes, or processors. The computers, processes, or processors are referred to as the *nodes* of the distributed system. (In the subsequent chapters we shall use a more technical notion, see Definition 2.6.) To be qualified as "autonomous", the nodes must at least be equipped with their own private control; thus, a parallel computer of the single-instruction, multiple-data (SIMD) model does not qualify as a distributed system. To be qualified as "interconnected", the nodes must be able to exchange information.

As (software) processes can play the role of nodes of a system, the definition includes software systems built as a collection of communicating processes, even when running on a single hardware installation. In most cases, however, a distributed system will at least contain several processors, interconnected by communication hardware.

More restrictive definitions of distributed systems are also found in the literature. Tanenbaum [Tan96], for example, considers a system to be distributed only if the existence of autonomous nodes is *transparent* to users of the system. A system distributed in this sense behaves like a virtual, stand-alone computer system, but the implementation of this transparency requires the development of intricate distributed control algorithms.

1.1.1 Motivation

Distributed computer systems may be preferred over sequential systems, or their use may simply be unavoidable, for various reasons, some of which are discussed below. This list is not meant to be exhaustive. The choice of a distributed system may be motivated by more than one of the arguments listed below, and some of the advantages may come as a spin-off after the choice has been made for another reason. The characteristics of a distributed system may vary also, depending on the reason for its existence, but this will be discussed in more detail in Subsections 1.1.2 through 1.1.6.

(1) *Information exchange.* The need to exchange data between different computers arose in the sixties, when most major universities and companies started to have their own mainframe computer. Cooperation between people of different organizations was facilitated by

the exchange of data between the computers of these organizations, and this gave rise to the development of so-called *wide-area networks* (WANs). ARPANET, the predecessor of the current Internet, went on-air in December, 1969. A computer installation connected in a wide-area network (sometimes called a *long-haul network*) is typically equipped with everything a user needs, such as backup storage, disks, many application programs, and printers.

Later computers became smaller and cheaper, and soon each single organization had a multitude of computers, nowadays often a computer for each person (a personal computer or workstation). In this case also the (electronic) exchange of information between personnel of one organization already required that the autonomous computers were connected. It is even not uncommon for a single person or family to have multiple computers in the home, and connect these in a small personal home-network.

(2) *Resource sharing.* Although with cheaper computers it became feasible to equip each employee of an organization with a private computer, the same is not the case for the peripherals (such as printers, backup storage, and disk units). On this smaller scale each computer may rely on dedicated servers to supply it with compilers and other application programs. Also, it is not effective to replicate all application programs and related file resources in all computers; apart from the waste of disk space, this would create unnecessary maintenance problems. So the computers may rely on dedicated nodes for printer and disk service. A network connecting computers on an organization-wide scale is referred to as a *local-area network* (LAN).

The reasons for an organization to install a network of small computers rather than a mainframe are cost reduction and extensibility. First, smaller computers have a better price–performance ratio than large computers; a typical mainframe computer may perform 50 times faster than a typical personal computer, but cost 500 times more. Second, if the capacity of a system is no longer sufficient, a network can be made to fit the organization's needs by adding more machines (file servers, printers, and workstations). If the capacity of a stand-alone system is no longer sufficient, replacement is the only option.

(3) *Increased reliability through replication.* Distributed systems have the potential to be more reliable than stand-alone systems because they have a *partial-failure* property. By this it is meant that some nodes of the system may fail, while others are still operating correctly and can

take over the tasks of the failed components. The failure of a stand-alone computer affects the entire system and there is no possibility of continuing the operation in this case. For this reason distributed architectures are a traditional concern in the design of highly reliable computer systems.

A highly reliable system typically consists of a two, three, or four times replicated uniprocessor that runs an application program and is supplemented with a voting mechanism to filter the outputs of the machines. The correct operation of a distributed system in the presence of failures of components requires rather complicated algorithmical support.

(4) *Increased performance through parallelization.* The presence of multiple processors in a distributed system opens up the possibility of decreasing the turn-around time for a computation-intensive job by splitting the job over several processors.

Parallel computers are designed specifically with this objective in mind, but users of a local-area network may also profit from parallelism by shunting tasks to other workstations.

(5) *Simplification of design through specialization.* The design of a computer system can be very complicated, especially if considerable functionality is required. The design can often be simplified by splitting the system into modules, each of which implements part of the functionality and communicates with the other modules.

On the level of a single program modularity is obtained by defining abstract data types and procedures for different tasks. A larger system may be defined as a collection of cooperating processes. In both cases, the modules may all be executed on a single computer. But it is also possible to have a local-area network with different types of computers, one equipped with dedicated hardware for number crunching, another with graphical hardware, a third with disks, etc.

1.1.2 Computer Networks

By a computer network we mean a collection of computers, connected by communication mechanisms by means of which the computers can exchange information. This exchange takes place by sending and receiving messages. Computer networks fit our definition of distributed systems. Depending on the distance between the computers and their ownership, computer networks are called either *wide-area networks* or *local-area networks.*

A wide-area network usually connects computers owned by different organizations (industries, universities, etc.). The physical distance between the nodes is typically 10 kilometers or more. Each node of such a network is a complete computer installation, including all the peripherals and a considerable amount of application software. The main object of a wide-area network is the exchange of information between users at the various nodes.

A local-area network usually connects computers owned by a single organization. The physical distance between the nodes is typically 10 kilometers or less. A node of such a network is typically a workstation, a file server, or a printer server, i.e., a relatively small station dedicated to a specific function within the organization. The main objects of a local-area network are usually information exchange and resource sharing.

The boundary between the two types of network cannot always be sharply drawn, and usually the distinction is not very important from the algorithmical viewpoint because similar algorithmical problems occur in all computer networks. Relevant differences with respect to the development of algorithms include the following.

(1) *Reliability parameters.* In wide-area networks the probability that something will go wrong during the transmission of a message can never be ignored; distributed algorithms for wide-area networks are usually designed to cope with this possibility. Local-area networks are much more reliable, and algorithms for them can be designed under the assumption that communication is completely reliable. In this case, however, the unlikely event in which something does go wrong may go undetected and cause the system to operate erroneously.

(2) *Communication time.* The message transmission times in wide-area networks are orders of magnitude larger than those in local-area networks. In wide-area networks, the time needed for processing a message can almost always be ignored when compared to the time of transmitting the message.

(3) *Homogeneity.* Even though in local-area networks not all nodes are necessarily equal, it is usually possible to agree on common software and protocols to be used within a single organization. In wide-area networks a variety of protocols is in use, which poses problems of conversion between different protocols and of designing software that is compatible with different standards.

(4) *Mutual trust.* Within a single organization all users may be trusted, but in a wide-area network this is certainly not the case. A wide-area

network requires the development of secure algorithms, safe against offensive users at other nodes.

Subsection 1.1.3 is devoted to a brief discussion of wide-area networks; local-area networks are discussed in Subsection 1.1.4.

1.1.3 Wide-area Networks

Historical development. Much pioneering work in the development of wide-area computer networks was done in projects of the Advanced Research Projects Agency (ARPA) of the United States Department of Defense. The network *ARPANET* became operational in 1969, and connected four nodes at that time. This network has grown to several hundreds of nodes, and other networks have been set up using similar technology (MILNET, CYPRESS). The ARPANET contains special nodes (called interface message processors (IMPs)) whose only purpose is to process the message traffic.

When UNIX[1] systems became widely used, it was recognized that there was a need for information exchange between different UNIX machines, to which end the **uucp** program (UNIX-to-UNIX CoPy) was written. With this program files could be exchanged via telephone lines and a networks of UNIX users, called the *UUCP network*, emerged rapidly. Yet another major network, called BITNET, was developed in the eighties because the ARPANET was owned by the Department of Defense, and only some organizations could connect to it.

Nowadays all these networks are interconnected; there exist nodes that belong to both (called *gateways*), allowing information to be exchanged between nodes of different networks. The introduction of a uniform address space and common protocols has turned the networks into a single virtual network, commonly known as the *Internet*. It differs from a "monolithic" network by having many owners and the lack of a single authoritative body, but its organizational diversity is carefully hidden from the users. The electronic address of the author (`gerard@cs.uu.nl`) provides no information about the network his department is connected to.

Organization and algorithmical problems. Wide-area networks are always organized as *point-to-point* networks. This means that communication between a pair of nodes takes place by a mechanism particular to these two nodes. Such a mechanism may be a telephone line, a fiber optic or satellite connection, etc. The interconnection structure of a point-to-point network

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

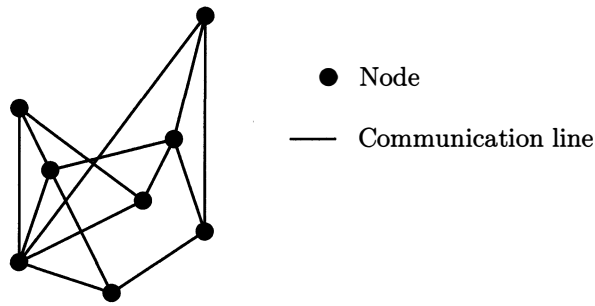

Figure 1.1 AN EXAMPLE OF A POINT-TO-POINT NETWORK.

can be conveniently depicted by drawing each node as a circle or box, and drawing a line between two nodes if there is a communication line between the two nodes; see Figure 1.1. In more technical language, the structure is represented by a *graph*, of which the edges represent the communication lines of the network. A summary of graph theory terminology is given in Appendix B.

The main purpose of wide-area networks is the exchange of information, for example in the form of electronic mail, bulletin boards, and remotely accessed files and databases. Most of these services are available through a single application, the web *browser*. The implementation of a suitable communication system for these purposes requires the solution of the following algorithmical problems, some of which are discussed in Part One of this book.

(1) *The reliability of point-to-point data exchange* (Chapter 3). Two nodes connected by a line exchange data through this line, but they must somehow cope with the possibility that the line is unreliable. Due to atmospheric noise, power dips, and other physical circumstances, a message sent through a line may be received with some parts garbled, or even lost. These transmission failures must be recognized and corrected.

This problem occurs not only for two nodes directly connected by a communication line, but also for nodes not directly connected, but communicating with the help of intermediate nodes. In this case the problem is even more complicated, because in addition messages may arrive in a different order from that in which they were sent, may arrive only after a very long period of time, or may be duplicated.

(2) *Selection of communication paths* (Chapter 4). In a point-to-point network it is usually too expensive to provide a communication line between each pair of nodes. Consequently, some pairs of nodes must rely on other nodes in order to communicate. The problem of *routing* concerns the selection of a path (or paths) between nodes that want to communicate. The algorithm used to select the path is related to the scheme by which nodes are named, i.e., the format of the address that one node must use to send a message to another node. Path selection in intermediate nodes is done using the address, and the selection can be done more efficiently if topological information is "coded" in the addresses.

(3) *Congestion control.* The throughput of a communication network may decrease dramatically if many messages are in transit simultaneously. Therefore the generation of messages by the various nodes must be controlled and made dependent on the available free capacity of the network. Some methods to avoid congestion are discussed in [Tan96, Section 5.3].

(4) *Deadlock prevention* (Chapter 5). Point-to-point networks are sometimes called *store-and-forward* networks, because a message that is sent via several intermediate nodes must be stored in each of these nodes, then forwarded to the next node. Because the memory space available for this purpose in the intermediate nodes is finite, the memory must be managed carefully in order to prevent deadlock situations. In such situations, there exists a set of messages, none of which can be forwarded because the memory of the next node on its route is fully occupied by other messages.

(5) *Security.* Networks connect computers having different owners, some of whom may attempt to abuse or even disrupt the installations of others. Since it is possible to log on a computer installation from anywhere in the world, reliable techniques for user authentication, cryptography, and scanning incoming information (e.g., for viruses) are required. Cryptographic methods (cf. [Tan96, Secion 7.1]) can be used to encrypt data for security against unauthorized reading and to implement electronic signatures for security against unauthorized writing.

1.1.4 Local-area Networks

A local-area network is used by an organization to connect a collection of computers it owns. Usually, the main purpose of this connection is to share

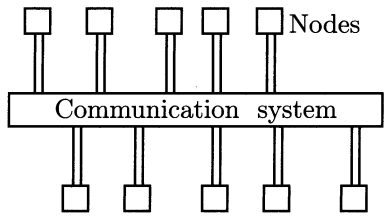

Figure 1.2 A BUS-CONNECTED NETWORK.

resources (files as well as hardware peripherals) and to facilitate the exchange of information between employees. Incidentally, networks are also used to speed up computations (by shunting tasks to other nodes) and to allow some nodes to be used as a stand-by for others in the event of their failure.

Examples and organization. In the first half of the nineteen-seventies, the Ethernet[2] local-area network was developed by the Xerox Corporation. While the names of wide-area networks, ARPANET, BITNET, etc., refer to specific networks, the names of local-area networks are usually product names. There was one ARPANET, one BITNET, and one UUCP net, and there is one Internet, but each company may set up its own private Ethernet, Token Ring, or SNA network.

Unlike wide-area networks, the Ethernet is organized using a *bus-like* structure, i.e., the communication between nodes takes place via a single mechanism to which all nodes are connected; see Figure 1.2. Bus-like organization has become very common for local-area networks, although there may be differences in what this mechanism looks like or how it is used.

The Ethernet design allows only one message to be transmitted at a time; other designs, such as the *token ring* (developed at the IBM Zürich Laboratory), allow for *spatial reuse*, which means that several messages can be transmitted via the communication mechanism at the same time. The bus organization requires little hardware and is therefore cheap, but it has the disadvantage that it is not a very *scalable* organization. This means that there is a fairly tight maximum to the number of nodes that can be connected by one bus. Large companies with many computers must connect them by several buses, and use *bridges* to connect the buses with each other, giving rise to a hierarchical overall network organization.

[2] Ethernet is a trademark of the Xerox Corporation.

Not all local-area networks use a bus organization. IBM has designed a point-to-point network product called SNA to allow its costumers to connect their various IBM products. The design of SNA was complicated by the requirement of making it compatible with each of the many networking products already offered by IBM.

Algorithmical problems. The implementation of local-area networks requires the solution of some, but not all, of the problems mentioned in the previous subsection on wide-area networks. Reliable data exchange is not so much of a problem because the buses are usually very reliable and fast. The routing problem does not occur in bus-like networks because each destination can be addressed directly via the bus. In ring-shaped networks all messages are usually sent in the same direction along the ring and removed by either the recipient or the sender, which also makes the routing problem practically void. There is no congestion in the bus because each message is received (taken from the bus) immediately after it is sent, but it is necessary to limit the congestion of messages waiting in the nodes to enter the bus. As messages are not stored in intermediate nodes, no store-and-forward deadlocks arise. Security mechanisms beyond the usual protection offered by an operating system are not necessary if all computers are owned by a single family or a single company that trusts its employees.

The use of local-area networks for the distributed execution of application programs (collections of processes, spread over the nodes of the network) requires the solution of the following distributed-control problems, some of which are discussed in Part Two.

(1) *Broadcasting and synchronization* (Chapter 6). If information must be made available to all processes, or all processes must wait until some global condition is satisfied, it is necessary to have a message-passing scheme that somehow "touches" all processes.
(2) *Election* (Chapter 7). Some tasks must be carried out by exactly one process of a set, for example, generating output or initializing a data structure. If, as is sometimes desirable or necessary, no process is assigned this task a priori, a distributed algorithm must be executed to select one of the processes to perform the task.
(3) *Termination detection* (Chapter 8). It is not always possible for the processes in a distributed system to observe directly that the distributed computation in which they are engaged has terminated. Termination detection is then necessary in order to make the computed results definitive.

(4) *Resource allocation.* A node may require access to some resource that is available elsewhere in the network, though it does not know in which node this resource is located. Maintaining a table that indicates the location of each resource is not always adequate, because the number of potential resources may be too large for this, or resources may migrate from one node to another. In such a case the requesting node may inquire at all or some other nodes about the availability of the resource, for example using broadcasting mechanisms. Algorithms for this problem can be based on the wave mechanisms described in Chapter 6; see, e.g., Baratz *et al.* [BGS87].

(5) *Mutual exclusion.* The problem of mutual exclusion arises if the processes must rely on a common resource that can be used only by one process at a time. Such a resource may be a printer device or a file that must be written. A distributed algorithm is then necessary to determine, if several processes request access simultaneously, which of them is allowed to use the resource first. Also it must be ensured that the next process begins to use the resource only after the previous process has finished using it.

(6) *Deadlock detection and resolution.* If processes must wait for each other (which is the case if they share resources, and also if their computation relies on data provided by other processes) a cyclic wait may occur, in which no further computation is possible. These deadlock situations must be detected and proper action must be undertaken to restart or continue the computation.

(7) *Distributed file maintenance.* When nodes place read and write requests for a remote file, these requests may be processed in an arbitrary order, and hence provision must be made to ensure that each node observes a consistent view of the file or files. Usually this is done by *time stamping* requests, as well as the information in files, and ordering incoming requests on the basis of their time stamps; see, e.g., [LL86].

1.1.5 Multiprocessor Computers

A multiprocessor computer is a computer installation that consists of several processors on a small scale, usually inside one large box. This type of computer system is distinguished from local-area networks by the following criteria. Its processors are homogeneous, i.e., they are identical as hardware. The geographical scale of the machine is very small, typically of the order of one meter or less. The processors are intended to be used together in

one computation (either to increase the speed or to increase the reliability). If the main design objective of the multiprocessor computer is to improve the speed of computation, it is often called a parallel computer. If its main design objective is to increase the reliability, it is often called a replicated system.

Parallel computers are classified into single-instruction, multiple-data (or SIMD) machines and multiple-instruction, multiple-data (or MIMD) machines. The SIMD machines have one instruction interpreter, but the instruction is carried out by a large number of arithmetic units. Clearly, these units lack the autonomy required in our definition of distributed systems, and therefore SIMD computers will not be considered in this book. The MIMD machines consist of several independent processors and they are classified as distributed systems.

The processors are usually equipped with dedicated hardware for communication with other processors. The communication between the processors can take place either via a bus or via point-to-point connections. If a bus organization is chosen, the architecture is scalable only to a certain limit.

A very popular processor for the design of multiprocessor computers was the Transputer[3] chip developed by Inmos; see Figure 1.3. A Transputer chip contains a central processing unit (CPU), a dedicated floating point unit (FPU), a local memory, and four dedicated communication processors. The chips lend themselves very well to the construction of networks of degree four (i.e., each node is connected to four other nodes). Inmos also produces special chips dedicated to communication, called *routers*. Each router can simultaneously handle the traffic of 32 Transputer links. Each incoming message is inspected to see via which link it must be forwarded; it is then sent via that link. The popularity of these machines peaked in the first half of the nineties.

Another example of a parallel computer is the Connection Machine CM-5[4] system, developed by Thinking Machines Corporation [LAD+92] and continued by Connection Machines Services, Inc. Each node of the machine consists of a fast processor and a vector processing unit, thus offering internal parallelism in addition to the parallelism offered by having many nodes. As each node has a potential performance of 128 million operations per second, and one machine may contain 16,384 nodes, the entire machine may execute over 10^{12} operations per second. (A maximal machine of 16,384 processors occupies a room of 900m^2 and is probably very expensive.) The

[3] Transputer is a trademark of Inmos, now SGS-Thomson Microelectronics.
[4] Connection Machine is a registered trademark and CM-5 a trademark of Thinking Machines Corporation.

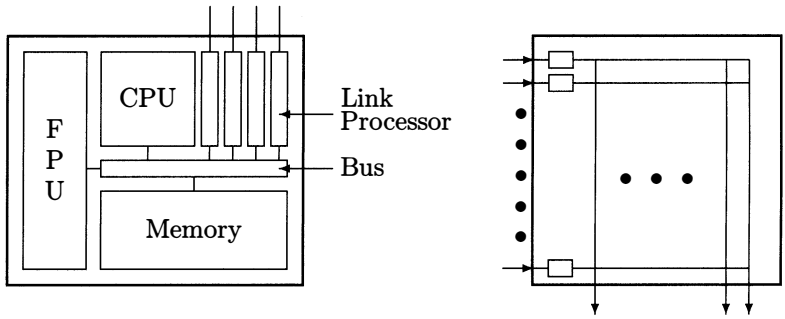

Figure 1.3 A TRANSPUTER AND A ROUTING CHIP.

nodes of the CM-5 are connected by three point-to-point communication networks. The *data network*, with a *fat tree* topology, is used to exchange data in a point-to-point manner between the processors. The *control network*, with a binary tree topology, performs dedicated operations such as global synchronization and combining inputs. The *diagnostic network* is invisible for the programmer and is used to propagate information about failed components. The computer can be programmed in both SIMD and (synchronous) MIMD mode.

In a parallel computer the computation is divided into subcomputations, each performed by one of the nodes. In a replicated system each node carries out the entire computation, after which the results are compared in order to detect and correct errors.

The construction of multiprocessor computers requires the solution of several algorithmical problems, some of which are similar to problems that arise in computer networks. Several of these problems are also discussed in this book.

(1) *Implementation of a message-passing system.* If the multiprocessor computer is organized as a point-to-point network a communication system must be designed. This poses problems similar to those that occur in the implementation of computer networks, such as transmission control, routing, and deadlock and congestion prevention. The solutions to these problems are often simpler than in the general case of computer networks. The routing problem, for example, is much simplified by regularity of the network's topology (i.e., a ring or grid) and the reliability of the nodes.

The Inmos C104 routing chips use a very simple routing algorithm

called *interval routing*, discussed in Subsection 4.4.2, which cannot be efficiently used in networks of arbitrary topology. This raises the question whether solutions for other problems, e.g., deadlock prevention, can be used in combination with this routing mechanism (see Project 5.5).

(2) *Implementation of a virtual shared memory.* Many parallel algorithms are designed for the so-called parallel random-access memory (PRAM) model, in which each processor has access to a shared memory. Architectures with a memory that is physically shared are not scalable; there is a firm limit to the number of processors that can be served by a single memory chip.

Research is therefore directed towards architectures that have several memory nodes, connected to the processors by an interconnection network.

(3) *Load balancing.* The computational power of a parallel computer is exploited only if the workload of a computation is spread uniformly over the processors; concentration of the work at a single node degrades the performance to that of a single node. If all steps of the computation can be determined at compile time, it is possible to distribute them statically. A more difficult case arises when units of work are created dynamically during a computation; in this case, intricate techniques are required. The task queues of the processors must be compared regularly, after which tasks must migrate from one processor to another. For an overview of some techniques and algorithms for load balancing, see Goscinski [Gos91, Chapter 9] or Harget and Johnson [HJ90].

(4) *Robustness against undetectable failures* (Part Three). In a replicated system there must be a mechanism to overcome failures in one or more processors. Of course, computer networks must also continue their operation in spite of the failure of a node, but there it is usually assumed that such a failure can be detected by other nodes (see, e.g., the Netchange algorithm in Section 4.3). The assumptions under which replicated systems must remain correct are much more severe since a processor may produce an erroneous answer but otherwise cooperate in the protocols exactly like a correctly operating processor. Voting mechanisms must be implemented to filter the results of the processors, so that only correct answers are delivered as long as a majority of the processors operates correctly.

1.1.6 Cooperating Processes

The design of complicated software systems may often be simplified by organizing the software as a collection of (sequential) processes, each with a well-defined, simple task.

A classical example to illustrate this simplification is *Conway's record conversion.* The problem is to read 80-character records and write the same data in 125-character records. After each input record an additional blank must be inserted, and each pair of asterisks ("**") must be replaced by an exclamation mark ("!"). Each output record must be followed by an end-of-record (EOR) character. The conversion can be carried out by a single program, but writing this program is very intricate. All functions, i.e., replacement of "**" by "!", the insertion of blanks, and the insertion of EOR characters, must be handled in a single loop.

The program is better structured as two cooperating processes. The first process, say p_1, reads the input cards and converts the input to a stream of printable characters, not divided into records. The second process, say p_2, receives the stream of characters and inserts an EOR after every 125 characters. A design as a collection of processes is usually assumed for operating systems, telephone switching centers, and, as will be seen in Subsection 1.2.1, communication software in computer networks.

A design as a collection of cooperating processes causes the application to be *logically* distributed, but it is quite possible to execute the processes on the same computer, in which case it is not *physically* distributed. It is of course the case that achieving physical distribution is easier for systems that are logically distributed. The operating system of a computer installation must control the concurrent execution of the processes and provide the means for communication and synchronization between the processes.

Processes that execute on the same processor have access to the same physical memory, hence it is most natural to use this memory for communication. One process writes in a certain memory location, and another process reads from this location. This model of *concurrent processes* was used by Dijkstra [Dij68] and Owicki and Gries [OG76]. Problems that have been considered in this context include the following.

(1) *Atomicity of memory operations.* It is frequently assumed that reading and writing on a single word of memory are atomic, i.e., the read or write executed by one process is completed before another read or write operation begins. If structures larger than a single word are to be updated, the operations must be carefully synchronized to avoid the reading of a partially updated structure. This can be done, for

example, by implementing *mutual exclusion* [Dij68] on the structure: while one process has access to the structure, no other process can start reading or writing. The implementation of mutual exclusion using shared variables is intricate, because of the possibility that several processes may seek entry to the structure at the same time.

The wait conditions imposed by mutually exclusive access to shared data may reduce the performance of the processes, for example if a "fast" process must wait for data currently accessed by a "slow" process. In recent years attention has concentrated on implementations of atomic shared variables that are *wait-free*, meaning that a process can read or write the data without waiting for any other process. Reading and writing may now overlap, but by careful design of the read and write algorithms, atomicity can be ensured. For an overview of algorithms for wait-free atomic shared variables, see Kirousis and Kranakis [KK89], or Attiya and Welch [AW98].

(2) *The producer–consumer problem.* Two processes, one of which writes in a shared buffer and the other of which reads from this buffer, must be coordinated to prevent the first process from writing when the buffer is full and the second process from reading when the buffer is empty. The producer–consumer problem arises when the solution to Conway's conversion problem is worked out; p_1 produces the intermediate stream of characters, and p_2 consumes them.

(3) *Garbage collection.* An application that is programmed using dynamic data structures may produce inaccessible memory cells, referred to as *garbage.* Formerly, an application had to be interrupted when the memory system ran out of free space, in order to allow a special program called the *garbage collector* to identify and reclaim inaccessible memory. Dijkstra *et al.* [DLM+78] proposed an *on-the-fly* garbage collector, which could run as a separate process concurrently with the application.

Intricate cooperation between the application and the collector is required, because the application may modify the pointer structure in the memory while the collector is deciding which cells are inaccessible. The algorithm must be carefully analyzed to show that the modifications do not cause accessible cells to be erroneously reclaimed by the mutator. An algorithm for on-the-fly garbage collection with a simpler correctness proof was proposed by Ben-Ari [BA84].

The solutions to the problems listed here demonstrate that very difficult problems of process interaction can be solved for processes that com-

municate via shared memory. However, the solutions are often extremely sophisticated and sometimes a very subtle interleaving of steps of different processes introduces erroneous results for solutions that seem correct at first (and second!) glance. Therefore, operating systems and programming languages offer primitives for a more structured organization of the interprocess communication.

(1) *Semaphores.* A semaphore [Dij68] is a non-negative (integer) variable whose value can be read and written in one atomic operation. A **V** operation increments its value, and a **P** operation decrements its value when it is positive (and suspends the process executing this operation as long as the value is zero).

Semaphores are an appropriate tool to implement mutual exclusion on a shared data structure: the semaphore is initialized to 1, and access to the structure is preceded by a **P** operation and followed by a **V** operation. Semaphores put a large responsibility for correct use onto each process; the integrity of shared data is violated if a process manipulates the data erroneously or does not execute the required **P** and **V** operations.

(2) *Monitors.* A monitor [Hoa74] consists of a data structure and a collection of procedures that can be executed on this data by calling processes in a mutually exclusive way. Because the data can be accessed solely via the procedures declared in the monitor, correct use of the data is ensured when the monitor is declared correctly. The monitor thus prevents unrestricted access to the data and synchronizes the access by different processes.

(3) *Pipes.* A pipe [Bou83] is a mechanism that moves a data stream from one process to another and synchronizes the two communicating processes; it is a pre-programmed solution to the producer–consumer problem.

The pipe is a basic communication mechanism in the UNIX operating system. If the program `p1` implements process p_1 of Conway's conversion problem and `p2` implements p_2, the UNIX command `p1 | p2` calls the two programs and connects them by a pipe. The output of `p1` is buffered and becomes the input of `p2`; `p1` is suspended when the buffer is full, and `p2` is suspended when the buffer is empty.

(4) *Message passing.* Some programming languages, such as occam and Ada, provide message passing as a mechanism for interprocess communication. Synchronization problems are relatively easy to solve using message passing; because a message cannot be received prior

to its sending, a temporal relation between events is induced by the exchange of a message.

Message passing can be implemented using monitors or pipes, and it is the natural means of communication for systems that run on distributed hardware (without shared memory). Indeed, the languages occam and Ada have been developed with physically distributed applications in mind.

1.2 Architecture and Languages

The software for implementing computer communication networks is very complicated. In this section it is explained how this software is usually structured in acyclically dependent modules called *layers* (Subsection 1.2.1). We discuss two network-architecture standards, namely, the ISO model of *Open Systems Interconnection*, a standard for wide-area networks, and the supplementary IEEE standard for local-area networks (Subsections 1.2.2 and 1.2.3). Also the languages used for programming distributed systems are briefly discussed (Subsection 1.2.4).

1.2.1 Architecture

The complexity of the tasks performed by the communication subsystem of a distributed system requires that this subsystem is designed in a highly structured way. To this end, networks are always organized as a collection of modules, each performing a very specific function and relying on services offered by other modules. In network organizations there is always a strict *hierarchy* between these modules, because each module exclusively uses the services offered by the previous module. The modules are called *layers* or *levels* in the context of network implementation; see Figure 1.4.

Each layer implements part of the functionality required for the implementation of the network and relies on the layer just below it. The services offered by layer i to layer $i+1$ are precisely described in the layer i to layer $i+1$ *interface* (briefly, the $i/(i+1)$ interface). When designing a network, the first thing to do is to define the number of layers and the interfaces between subsequent layers.

The functionality of each layer must be implemented by a distributed algorithm, such that the algorithm for layer i solves a "problem" defined by the $i/(i+1)$ interface, under the "assumptions" defined in the $(i-1)/i$ interface. For example, the $(i-1)/i$ interface may specify that messages are transported from node p to node q, but some messages may be lost,

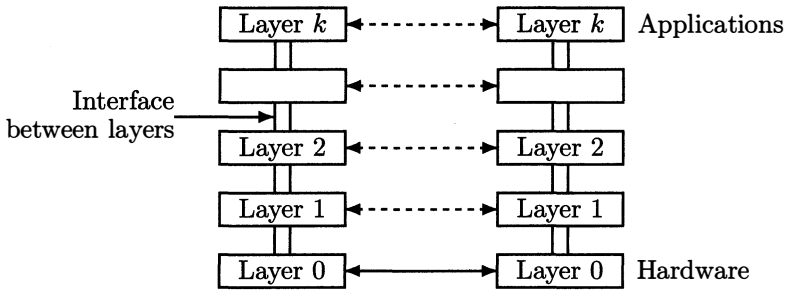

Figure 1.4 A LAYERED NETWORK ARCHITECTURE.

while the $i/(i+1)$ interface specifies that messages are transmitted from p to q reliably. The algorithmic problem for layer i is then to implement reliable message passing using unreliable message passing, which is usually done using acknowledgements and the retransmitting of lost messages (see Subsection 1.3.1 and Chapter 3). The solution of this problem defines the type of messages exchanged by the layer-i processes and the meaning of these messages, i.e., how the processes should react to these messages. The rules and conventions used in the "conversation" between the processes at layer i are referred to as the *layer-i protocol.*

The lowest layer of the hierarchy (layer 0 in Figure 1.4) is always the hardware layer. The 0/1 interface describes the procedures by which layer 1 can transmit raw information via the connecting wires, and a definition of the level itself specifies what types of wire are used, how many volts represents a one or a zero, etc. An important observation is that a change in the implementation of layer 0 (the replacement of wires by other wires or satellite connections) does not require the 0/1 interface to be changed. The same holds at higher layers: the layer interfaces serve to screen the implementation of a layer from other layers, and the implementation can be changed without affecting the other layers.

By the network *architecture* we mean the collection of layers and the accompanying definitions of all interfaces and protocols. As a network may contain nodes produced by various manufacturers, programmed with software written by different companies, it is important that products of different companies are compatible. The importance of compatibility has been recognized worldwide and therefore standard network architectures have been developed. In the next subsection two standards are discussed that have received an "official" status because they are adopted by influential

organizations (the International Standards Organization, ISO, and the Institute of Electrical and Electronic Engineers, IEEE).

The Transmission control protocol/Internet protocol (TCP/IP) is a collection of protocols used in the Internet. TCP/IP is not an official standard, but is used so widely that is has become a *de facto* standard. The TCP/IP protocol family (see Davidson [Dav88] for an introduction) is structured according to the layers of the OSI model discussed in the next subsection, but the protocols can be used in wide-area as well as in local-area networks. The higher layers contain protocols for electronic mail (the simple mail-transfer protocol, SMTP), file transfer (the file-transfer protocol, FTP), and bidirectional communication for remote login (Telnet).

1.2.2 The OSI Reference Model

The International Standards Organization (ISO) has fixed a standard for computer networking products such as those used (mainly) in wide-area networks. Their standard for network architectures is called the *Open-Systems Interconnection* (OSI) reference model, and will be described briefly in this subsection. Because the standard is not entirely appropriate for use in local-area networks, additional IEEE standards for local-area networks are discussed in the next subsection.

The OSI reference model consists of seven layers, namely the *physical*, *data-link*, *network*, *transport*, *session*, *presentation*, and *application* layers. The reference model specifies the interfaces between the layers and provides, for each layer, one or more standard protocols (distributed algorithms to implement the layer).

The physical layer (1). The purpose of the physical layer is to transmit sequences of bits over a communication channel. As the name of the layer suggests, this goal is achieved by means of a physical connection between two nodes, such as a telephone line, fiber optic connection, or satellite connection. The design of the layer itself is purely a matter for electrical engineers, while the 1/2 interface specifies the procedures by which the next layer calls the services of the physical layer. The service of the physical layer is not reliable; the bitstream may be scrambled during its transmission.

The data-link layer (2). The purpose of the data-link layer is to mask the unreliability of the physical layer, that is, to provide a reliable link to the higher layers. The data-link layer only implements a reliable connection between nodes that are directly connected by a physical link, because it

is built directly upon the physical layer. (Communication between non-adjacent nodes is implemented in the network layer.)

To achieve its goal, the layer divides the bitstream into pieces of fixed length, called *frames*. The receiver of a frame can check whether the frame was received correctly by verifying its *checksum*, which is some redundant information added to each frame. There is a feedback from the receiver to the sender to inform the sender about correctly or incorrectly received frames; this feedback takes place by means of *acknowledgement* messages. The sender will send a frame anew if it turns out that it is received incorrectly or completely lost.

The general principles explained in the previous paragraph can be refined to a variety of different data-link protocols. For example, an acknowledgement message may be sent for frames that are received (positive acknowledgements) or for frames that are missing from the collection of received frames (negative acknowledgements). The ultimate responsibility for the correct transmission of all frames may be at the sender's or the receiver's side. Acknowledgements may be sent for single frames or blocks of frames, the frames may have sequence numbers or not, etc.

The network layer (3). The purpose of the network layer is to provide a means of communication between all pairs of nodes, not just those connected by a physical channel. This layer must select the routes through the network used for communication between non-adjacent nodes and must control the traffic load of each node and channel.

The selection of routes is usually based on information about the network topology contained in *routing tables* stored in each node. The network layer contains algorithms to update the routing tables if the topology of the network changes (owing to a node or channel failure or recovery). Such a failure or recovery is detected by the data-link layer.

Although the data-link layer provides reliable service to the network layer, the service offered by the network layer is not reliable. Messages (called *packets* in this layer) sent from one node to the other may follow different paths, causing one message to overtake the other. Owing to node failures messages may be lost (a node may go down while holding a message), and owing to superfluous retransmissions messages may even be duplicated. The layer may guarantee a bounded packet lifetime; i.e., there exists a constant c such that each packet is either delivered at the destination node within c seconds, or lost.

The transport layer (4). The purpose of the transport layer is to mask the unreliability introduced by the network layer, i.e., to provide reliable *end-to-end* communication between any two nodes. The problem would be similar to the one solved by the data-link layer, but it is complicated by the possibility of the duplication and reordering of messages. This makes it impossible to use cyclic sequence numbers, unless a bound on the packet lifetime is guaranteed by the network layer.

The algorithms used for transmission control in the transport layer use similar techniques to the algorithms in the data-link layer: sequence numbers, feedback via acknowledgements, and retransmissions.

The session layer (5). The purpose of the session layer is to provide facilities for maintaining connections between processes at different nodes. A connection can be opened and closed and between opening and closing the connection can be used for data exchange, using a session address rather than repeating the address of the remote process with each message. The session layer uses the reliable end-to-end communication offered by the transport layer, but structures the exchanged messages into sessions.

A session can be used for file transfer or remote login. The session layer can provide the mechanisms for recovery if a node crashes during a session and for mutual exclusion if critical operations may not be performed at both ends simultaneously.

The presentation layer (6). The purpose of the presentation layer is to perform data conversion where the representation of information in one node differs from the representation in another node or is not suitable for transmission. Below this layer (i.e., at the 5/6 interface) data is in transmittable and standardized form, while above this layer (i.e., at the 6/7 interface) data is in user- or computer-specific form. The layer performs data *compression* and *decompression* to reduce the amount of data handed via the lower layers. The layer performs data *encryption* and *decryption* to ensure its confidentiality and integrity in the presence of malicious parties that aim to receive or corrupt the transmitted data.

The application layer (7). The purpose of the application layer is to fulfill concrete user requirements such as file transmission, electronic mail, bulletin boards, or virtual terminals. The wide variety of possible applications makes it impossible to standardize the complete functionality of this layer, but for some of the applications listed here standards have been proposed.

1.2.3 The OSI Model in Local-area Networks: IEEE Standards

The design of the OSI reference model is influenced to a large extent by the architectures of existing wide-area networks. The technology used in local-area networks poses different software requirements, and due to this some of the layers may be almost absent in local-area networks. If the network organization relies on a common bus shared by all nodes (see Subsection 1.1.4), the network layer is almost empty because each pair of nodes is connected directly via the bus. The design of the transport layer is very much simplified by the limited amount of non-determinism introduced by the bus, as compared to an intermediate point-to-point network. In contrast, the data-link layer is complicated by the fact that the same physical medium is accessed by a potentially large number of nodes.

In answer to these problems the IEEE has approved additional standards, covering only the lower levels of the OSI hierarchy, to be used in local-area networks (or, to be more precise, all networks that are bus-structured rather than point-to-point). Because no single standard could be general enough to encompass all networks already widely in use, the IEEE has approved three different, non-compatible standards, namely CSMA/CD, token bus, and token ring. The data-link layer is replaced by two sublayers, namely the *medium access control* and the *logical link control* sublayers.

The physical layer (1). The purpose of the physical layer in the IEEE standards is similar to that of the original ISO standard, namely to transmit sequences of bits. The actual standard definitions (the type of wiring, etc.) are, however, radically different, due to the fact that all communication takes place via a commonly accessed medium rather than point-to-point connections.

The medium access control sublayer (2a). The purpose of this sublayer is to resolve conflicts that arise between nodes that want to use the shared communication medium. A statical approach would once and for all schedule the time intervals during which each node is allowed to use the medium. This method wastes a lot of bandwidth, however, if only a few nodes have data to transmit, and all other nodes are silent; the medium remains idle during the times scheduled for the silent nodes.

In token buses and token rings access to the medium is on a round-robin basis: the nodes circulate a privilege, called the *token*, among them and the node holding this token is allowed to use the medium. If the node holding the token has no data to transmit, it passes the token to the next node. In a token ring the cyclic order in which nodes get their turn is determined by

the physical connection topology (which is, indeed, a ring), while in a token bus the cyclic order is determined dynamically based on the order of the node addresses.

In the carrier sense multiple access with collision detection (CSMA/CD) standard, nodes observe when the medium is idle, and if so they are allowed to send. If two or more nodes start to send (approximately) simultaneously, there is a *collision*, which is detected and causes each node to interrupt its transmission and try again at a later time.

The logical link control sublayer (2b). The purpose of this layer is comparable to the purpose of the data-link layer in the OSI model, namely to control the exchange of data between nodes. The layer provides error control and flow control, using techniques similar to those used in the OSI protocols, namely sequence numbers and acknowledgements.

Seen from the viewpoint of the higher layers, the logical link control sublayer appears like the network layer of the OSI model. Indeed, communication between any pair of nodes takes place without using intermediate nodes, and can be handled directly by the logical link control sublayer. A separate network layer is therefore not implemented in local-area networks; instead, the transport layer is built directly on top of the logical link control sublayer.

1.2.4 Language Support

The implementation of one of the software layers of a communication network or a distributed application requires that the distributed algorithm used in that layer or application is coded in a programming language. The actual coding is of course highly influenced by the language and especially by the primitives it offers. Because in this book we concentrate on the algorithms and not on their coding as a program, our basic model of processes is based on process states and state transitions (see Subsection 2.1.2) and not on the execution of instructions taken from a prescribed set. Of course it is inevitable that where we present algorithms, some formal notation is required; the programming notation used in this book is presented in Appendix A.

In this subsection we describe some of the constructs one may observe in actual programming languages designed for distributed systems. We confine ourselves here to a brief description of these constructs; for more details and examples of actual languages that use the various constructs, see, e.g., Bal [Bal90]. A language for programming distributed applications must provide

the means to express parallelism, process interaction, and non-determinism. Parallelism is of course required to program the various nodes of the system in such a way that the nodes will execute their part of the program concurrently. Communication between the nodes must also be supported by the programming language. Non-determinism is necessary because a node must sometimes be able to receive a message from different nodes, or be able to either send or receive a message.

Parallelism. The most appropriate degree of parallelism in a distributed application depends on the ratio between the cost of communication and the cost of computation. A larger degree of parallelism allows for faster execution, but also requires more communication, so if communication is expensive, the gain in computation speed may be lost in additional communication cost.

Parallelism is usually expressed by defining several *processes*, each being a sequential entity with its own state space. A language may either offer the possibility of statically defining a collection of processes or allow the dynamic creation and termination of processes. It is also possible to express parallelism by means of parallel statements or in a functional programming language. Parallelism is not always explicit in a language; the partitioning of code into parallel processes may be performed by a sophisticated compiler.

Communication. Communication between processes is inherent to distributed systems: if processes do not communicate, each process operates in isolation from other processes and should be studied in isolation, not as part of a distributed system. When processes cooperate in a computation, communication is necessary if one process needs an intermediate result produced by another process. Also, synchronization is necessary, because the former process must be suspended until the result is available. *Message passing* achieves both communication and synchronization; a *shared memory* achieves only communication: additional care must be taken to synchronize processes that communicate using shared memory.

In languages that provide message passing, "send" and "receive" operations are available. Communication takes place by the execution of the send operation in one process (therefore called the sender process) and the receive operation in another process (the receiver process). The arguments of the send operation are the receiver's address and additional data, forming the content of the message. This additional data becomes available to the receiver when the receive statement is executed, i.e., this implements the communication. The receive operation can be completed only after the

send operation has been executed, which implements the synchronization. In some languages a receive operation is not available explicitly; instead, a procedure or operation is activated implicitly when a message is received.

A language may provide *synchronous* message passing, in which case the send operation is completed only after execution of the receive operation. In other words, the sender is blocked until the message has been received, and a two-way synchronization between sender and receiver results.

Messages can be sent point-to-point, i.e., from one sender to one receiver, or broadcast, in which case the same message is received by all receivers. The term multicast is also used to refer to messages that are sent to a collection of (not necessarily all) processes. A somewhat more structured communication primitive is the *remote procedure call* (RPC). To communicate with process b, process a calls a procedure present in process b by sending the parameters of the procedure in a message; a is suspended until the result of the procedure is returned in another message.

An alternative for message passing is the use of *shared memory* for communication; one process writes a value to a variable, and another process reads the value. Synchronization between the processes is harder to achieve, because reading a variable can be done before the variable has been written. Using synchronization primitives such as *semaphores* [Dij68] or *monitors* [Hoa78], it is possible to implement message passing in a shared-variable environment. Conversely, it is also possible to implement a (virtual) shared memory in a message-passing environment, but this is very inefficient.

Non-determinism. At many points in its execution a process may be able to continue in various different ways. A receive operation is often non-deterministic because it allows the receipt of messages from different senders. Additional ways to express non-determinism are based on *guarded commands*. A guarded command in its most general form is a list of statements, each preceded by a boolean expression (its guard). The process may continue its execution with any of the statements for which the corresponding guard evaluates to *true*. A guard may contain a receive operation, in which case it evaluates to true if there is a message available to be received.

1.3 Distributed Algorithms

The previous sections have given reasons for the use of distributed computer systems and explained the nature of these systems; the need to program these systems arises as a consequence. The programming of distributed systems must be based on the use of correct, flexible, and efficient algorithms.

In this section it is argued that the development of distributed algorithms is a craft quite different in nature from the craft used in the development of centralized algorithms. Distributed and centralized systems differ in a number of essential respects, treated in Subsection 1.3.1 and exemplified in Subsection 1.3.2. Distributed algorithms research has therefore developed as an independent field of scientific research; see Subsection 1.3.3. This book is intended to introduce the reader to this research field. The objectives of the book and a selection of results included in the book are stated in Subsection 1.4.

1.3.1 Distributed versus Centralized Algorithms

Distributed systems differ from centralized (uniprocessor) computer systems in three essential respects, which we now discuss.

(1) *Lack of knowledge of global state.* In a centralized algorithm control decisions can be made based upon an observation of the state of the system. Even though the entire state usually cannot be accessed in a single machine operation, a program may inspect the variables one by one, and make a decision after all relevant information has been considered. No data is modified between the inspection and the decision, and this guarantees the integrity of the decision.

Nodes in a distributed system have access only to their own state and not to the global state of the entire system. Consequently, it is not possible to make a control decision based upon the global state. It is the case that a node may receive information about the state of other nodes and base its control decisions upon this information. In contrast with centralized systems, the fact that the received information is old may render the information invalid, because the state of the other node may have changed between the sending of the state information and the decision based upon it.

The state of the communication subsystem (i.e., what messages are in transit in it at a certain moment) is never directly observed by the nodes. This information can only be deduced indirectly by comparing information about messages sent and received by the nodes.

(2) *Lack of a global time-frame.* The events constituting the execution of a centralized algorithm are totally ordered in a natural way by their temporal occurrence; for each pair of events, one occurs earlier or later than the other. The temporal relation induced on the events constituting the execution of a distributed algorithm is not total; for

some pairs of events there may be a reason for deciding that one occurs before the other, but for other pairs it is the case that neither of the events occurs before the other [Lam78].

Mutual exclusion can be achieved in a centralized system by requiring that if the access of process p to the resource starts later than the access of process q, then the access of process p must start after the access of process q has ended. Indeed, all such events (the starting and ending of the access of processes p and q) are totally ordered by the temporal relation; in a distributed system they are not, and the same strategy is not sufficient. Processes p and q may start accessing the resource, while the start of neither temporally precedes the start of the other.

(3) *Non-determinism.* A centralized program may describe the computation as it unrolls from a certain input unambiguously; given the program and the input, only a single computation is possible. In contrast, the execution of a distributed system is usually non-deterministic, due to possible differences in execution speed of the system components.

Consider the situation where a server process may receive requests from an unknown number of client processes. The server cannot suspend the processing of requests until all requests have been received, because it is not known how many messages will arrive. Consequently, each request must be processed immediately, and the order of processing is the order in which the requests arrive. The order in which the clients send their requests may be known, but as the transmission delays are not known, the requests may arrive in a different order.

The combination of lack of knowledge of the global state, lack of a global time-frame, and non-determinism makes the design of distributed algorithms an intricate craft, because the three aspects interfere in several ways.

The concepts of *time* and *state* are highly related; in centralized systems the notion of time may be defined by considering the *sequence* of states assumed by the system during its execution. Even though in a distributed system a global state can be defined, and an execution can be seen as a sequence of global states (cf. Definition 2.2), this view is of limited use since the execution can also be described by other sequences of global states (Theorem 2.21). Those alternative sequences usually consist of different global states; this gives the statement "the system assumed this or that state during its execution" a very dubious meaning.

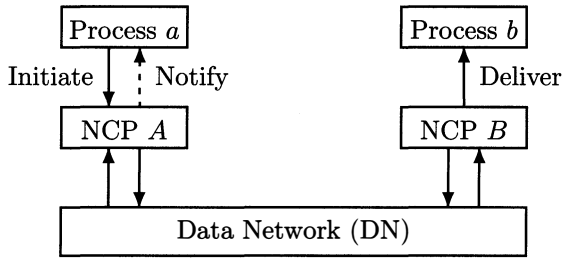

Figure 1.5 A SIMPLIFIED NETWORK ARCHITECTURE.

Lack of knowledge of the global state could be compensated for if it were possible to predict this global state from the algorithm that is being executed. Unfortunately, this is not possible due to the inherent nondeterminism in the execution of distributed systems.

1.3.2 An Example: Single-message Communication

We shall illustrate the difficulties imposed by the lack of knowledge of the global state and the lack of a global time-frame by an example problem, discussed by Belsnes [Bel76], namely the reliable exchange of information via an unreliable medium. Consider two processes, a and b, connected by a data network, which transmits messages from one process to the other. A message may be received an arbitrarily long time after it is sent, but may also be lost altogether in the network. The reliability of the communication is increased by the use of network control procedures (NCPs), via which a and b access the network. Process a initiates the communication by giving an information unit m to NCP A. The interaction between the NCPs (via the data network, DN) must ensure that the information m is delivered at process b (by NCP B), after which a is notified of the delivery (by NCP A). The communication structure is depicted in Figure 1.5.

Even if only a single information unit is to be transported from a to b, the network's unreliability forces NCP A and NCP B to engage in a conversation consisting of several messages. They maintain state information about this conversation, but because the number of possible conversation partners for each process is large, it is required that state information is discarded after the message exchange is complete. The initialization of status information is called *opening* and its discarding is called *closing* the conversation. Observe

that, after closing the conversation, an NCP is in exactly the same state as before opening it; this is called the *closed* state.

Information unit m is said to be *lost* if a was notified of its receipt by b, but the unit was never actually delivered to b. Unit m is said to be *duplicated* if it was delivered twice. Reliable communication mechanisms prevent losses as well as duplications. It is assumed that NCPs may fail (crash), after which they are restarted in the closed state (effectively losing all information about a currently open conversation).

No reliable communication is achievable. As a first observation, it can be shown that no matter how intricately the NCPs are designed, it is not possible to achieve completely reliable communication. This observation can be made independently of the design of the data network or the NCPs and only relies on the assumption that an NCP may lose information about an active conversation.

To see this, assume that after initialization of a communication by a, NCP A and NCP B start a conversation, during which NCP B is supposed to deliver m to b after the receipt of a message M from NCP A. Consider the case where NCP B crashes and is restarted in the closed state after NCP A has sent message M. In this situation, neither NCP A nor NCP B can tell whether m was already delivered when NCP B crashed; NCP A because it cannot observe the events in NCP B (lack of knowledge of the global state) and NCP B because it has crashed and was restarted in the closed state. Regardless of how the NCPs continue their conversation, an error may be introduced. If NCP A sends the message to NCP B again and NCP B delivers the message, a duplication may arise. If notification to a is given without delivery, a loss may arise.

We shall now evaluate several possible designs for the NCPs with respect to the possibility of losing or duplicating messages. We shall try to design the protocols in such a way that losses are avoided in any case.

A one-message conversation. In the simplest possible design, NCP A sends the data unaltered via the network, notifies a, and closes, in a single action upon initialization. NCP B always delivers a message it receives to b and closes after each delivery.

This protocol introduces a loss whenever the network refuses to deliver a message, but there is no possibility of introducing duplications.

A two-message conversation. A limited protection against loss of messages is offered by the addition of acknowledgements to the protocol. In a

normal conversation, NCP A sends a data message $\langle \mathbf{data}, m \rangle$ and waits for the receipt of an acknowledgement message $\langle \mathbf{ack} \rangle$ from NCP B. When this message is received, NCP A closes the conversation. NCP B, upon receipt of the $\langle \mathbf{data}, m \rangle$ message, delivers m to b, replies with an $\langle \mathbf{ack} \rangle$ message, and closes. Summarizing, an error-free conversation consists of three events.

1. NCP A send $\langle \mathbf{data}, m \rangle$
2. NCP B receive $\langle \mathbf{data}, m \rangle$, deliver m, send $\langle \mathbf{ack} \rangle$, close
3. NCP A receive $\langle \mathbf{ack} \rangle$, notify, close

The possibility of loss of a data message forces NCP A to send $\langle \mathbf{data}, m \rangle$ again if the acknowledgement is not received after some time. (Due to lack of knowledge of the global state, NCP A cannot observe whether $\langle \mathbf{data}, m \rangle$ was lost, $\langle \mathbf{ack} \rangle$ was lost, or NCP B crashed between receiving $\langle \mathbf{data}, m \rangle$ and sending $\langle \mathbf{ack} \rangle$.) To this end, NCP A awaits the receipt of an acknowledgement for a limited amount of time, and if no such a message is received, a timer runs out and a *timeout* occurs. It can be easily seen that this option of retransmission introduces the possibility of a duplicate, namely, if not the original data message, but its acknowledgement, was lost, as in the following scenario.

1. NCP A send $\langle \mathbf{data}, m \rangle$
2. NCP B receive $\langle \mathbf{data}, m \rangle$, deliver m, send $\langle \mathbf{ack} \rangle$, close
3. DN $\langle \mathbf{ack} \rangle$ is lost
4. NCP A timeout, send $\langle \mathbf{data}, m \rangle$
5. NCP B receive $\langle \mathbf{data}, m \rangle$, deliver m, send $\langle \mathbf{ack} \rangle$, close
6. NCP A receive $\langle \mathbf{ack} \rangle$, notify, close

But not only do acknowledgements introduce the possibility of duplicates, they also fail to safeguard against losses, as the following scenario shows. Process a offers two information units, m_1 and m_2, for transmission.

1. NCP A send $\langle \mathbf{data}, m_1 \rangle$
2. NCP B receive $\langle \mathbf{data}, m_1 \rangle$, deliver m_1, send $\langle \mathbf{ack} \rangle$, close
3. NCP A timeout, send $\langle \mathbf{data}, m_1 \rangle$
4. NCP B receive $\langle \mathbf{data}, m_1 \rangle$, deliver m_1, send $\langle \mathbf{ack} \rangle$, close
5. NCP A receive $\langle \mathbf{ack} \rangle$, notify, close
6. NCP A send $\langle \mathbf{data}, m_2 \rangle$
7. DN $\langle \mathbf{data}, m_2 \rangle$ is lost
8. NCP A receive $\langle \mathbf{ack} \rangle$ (step 2), notify, close

Message m_1 is duplicated as in the previous scenario, but the first acknowledgement was delivered slowly instead of lost, causing a loss of a later in-

formation unit. The slow delivery is not detected due to the lack of a global time.

The problem of reliable interprocess communication can be solved more easily if a weak notion of global time is assumed, namely, that there exists an upper bound T on the transmission delay of any message sent through the network. This is considered a *global* timing assumption, because it induces a temporal relation between events in different nodes (namely, the sending by NCP A and a receipt by NCP B). The receipt of messages from earlier conversations can be prevented in this protocol by closing the conversation in NCP A only $2T$ after sending the last message.

A three-message conversation. As the two-message protocol loses or duplicates an information unit when an acknowledgement is lost or delayed, one may consider adding a third message to the conversation, informing NCP B that NCP A has received the acknowledgement. A normal conversation then consists of the following events.

1. NCP A send $\langle$ **data**, m $\rangle$
2. NCP B receive $\langle$ **data**, m $\rangle$, deliver m, send $\langle$ **ack** $\rangle$
3. NCP A receive $\langle$ **ack** $\rangle$, notify, send $\langle$ **close** $\rangle$, close
4. NCP B receive $\langle$ **close** $\rangle$, close

A loss of the $\langle$ **data**, m $\rangle$ message causes a timeout in NCP A, in which case NCP A retransmits the message. A loss of the $\langle$ **ack** $\rangle$ message also causes a retransmission of $\langle$ **data**, m $\rangle$, but this does not lead to a duplication because NCP B has an open conversation and recognizes the message it has already received.

Unfortunately, the protocol may still lose and duplicate information. Because NCP B must be able to close even when the $\langle$ **close** $\rangle$ message is lost, NCP B must retransmit the $\langle$ **ack** $\rangle$ message if it receives no $\langle$ **close** $\rangle$ message. NCP A replies saying that it has no conversation (a $\langle$ **nocon** $\rangle$ message), after which NCP B closes. The retransmission of $\langle$ **ack** $\rangle$ may arrive, however, in the next conversation of NCP A and be interpreted as an acknowledgement in that conversation, causing the next information unit to be lost, as in the following scenario.

1.	NCP A	send $\langle$ **data**, m_1 $\rangle$
2.	NCP B	receive $\langle$ **data**, m_1 $\rangle$, deliver m_1, send $\langle$ **ack** $\rangle$
3.	NCP A	receive $\langle$ **ack** $\rangle$, notify, send $\langle$ **close** $\rangle$, close
4.	DN	$\langle$ **close** $\rangle$ is lost
5.	NCP A	send $\langle$ **data**, m_2 $\rangle$
6.	DN	$\langle$ **data**, m_2 $\rangle$ is lost
7.	NCP B	retransmit $\langle$ **ack** $\rangle$ (step 2)
8.	NCP A	receive $\langle$ **ack** $\rangle$, notify, send $\langle$ **close** $\rangle$, close
9.	NCP B	receive $\langle$ **close** $\rangle$, close

Again the problem has arisen because messages of one conversation have interfered with another conversation. This can be ruled out by selection of a pair of new conversation identification numbers for each new conversation, one by NCP A and one by NCP B. The numbers chosen are included in all messages of the conversation, and are used to verify that a received message indeed belongs to the current conversation. The normal conversation of the three-message protocol is as follows.

1.	NCP A	send $\langle$ **data**, m, x $\rangle$
2.	NCP B	receive $\langle$ **data**, m, x $\rangle$, deliver m, send $\langle$ **ack**, x, y $\rangle$
3.	NCP A	receive $\langle$ **ack**, x, y $\rangle$, notify, send $\langle$ **close**, x, y $\rangle$, close
4.	NCP B	receive $\langle$ **close**, x, y $\rangle$, close

This modification of the three-message protocol excludes the erroneous conversation given earlier, because the message received by NCP A in step 8 is not accepted as an acknowledgement for the data message sent in step 5. However, NCP B does not verify the validity of a $\langle$ **data**, m, x $\rangle$ before delivering m (in step 2), which easily leads to duplication of information. If the message sent in step 1 is delayed and retransmitted, a later-arriving $\langle$ **data**, m, x $\rangle$ message causes NCP B to deliver information m again.

Of course, NCP B should also verify the validity of messages it receives before delivering the data. We consider a modification of the three-message conversation in which NCP B delivers the data in step 4 rather than in step 2. Notification is now given by NCP A *before* delivery by NCP B, but because NCP B has already received the information this seems justified. It must be ensured, though, that NCP B will now deliver the data in any case; in particular, when the $\langle$ **close**, x, y $\rangle$ message is lost. NCP B repeats the $\langle$ **ack**, x, y $\rangle$ message, to which NCP A replies with a $\langle$ **nocon**, x, y $\rangle$ message, causing NCP B to deliver and close, as in the following scenario.

1. NCP A send $\langle$ **data**, m, x $\rangle$
2. NCP B receive $\langle$ **data**, m, x $\rangle$, send $\langle$ **ack**, x, y $\rangle$
3. NCP A receive $\langle$ **ack**, x, y $\rangle$, notify, send $\langle$ **close**, x, y $\rangle$, close
4. DN $\langle$ **close**, x, y $\rangle$ is lost
5. NCP B timeout, retransmit $\langle$ **ack**, x, y $\rangle$
6. NCP A receive $\langle$ **ack**, x, y $\rangle$, reply $\langle$ **nocon**, x, y $\rangle$
7. NCP B receive $\langle$ **nocon**, x, y $\rangle$, deliver m, close

It turns out that, in order to avoid loss of information, NCP B must deliver the data even if NCP A does not confirm having a connection with identifications x and y. This renders the validation mechanism useless for NCP B, leading to the possibility of duplication of information as in the following scenario.

1. NCP A send $\langle$ **data**, m, x $\rangle$
2. NCP A timeout, retransmit $\langle$ **data**, m, x $\rangle$
3. NCP B receive $\langle$ **data**, m, x $\rangle$ (sent in step 2), send $\langle$ **ack**, x, y_1 $\rangle$
4. NCP A receive $\langle$ **ack**, x, y_1 $\rangle$, notify, send $\langle$ **close**, x, y_1 $\rangle$, close
5. NCP B receive $\langle$ **close**, x, y_1 $\rangle$, deliver m, close
6. NCP B receive $\langle$ **data**, m, x $\rangle$ (sent in step 1), send $\langle$ **ack**, x, y_2 $\rangle$
7. NCP A receive $\langle$ **ack**, x, y_2 $\rangle$, reply $\langle$ **nocon**, x, y_2 $\rangle$
8. NCP B receive $\langle$ **nocon**, x, y_2 $\rangle$ in reply to $\langle$ **ack**, x, y_2 $\rangle$, deliver m, close

A four-message conversation. The delivery of information from old conversations can be avoided by having the NCPs mutually agree upon their conversation identification numbers before any data is delivered, as in the following conversation.

1. NCP A send $\langle$ **data**, m, x $\rangle$
2. NCP B receive $\langle$ **data**, m, x $\rangle$, send $\langle$ **open**, x, y $\rangle$
3. NCP A receive $\langle$ **open**, x, y $\rangle$, send $\langle$ **agree**, x, y $\rangle$
4. NCP B receive $\langle$ **agree**, x, y $\rangle$, deliver m, send $\langle$ **ack**, x, y $\rangle$, close
5. NCP A receive $\langle$ **ack**, x, y $\rangle$, notify, close

The possibility of a crash of NCP B forces the error handling to be such that a duplicate may still occur, even when no NCP actually crashes. An error message $\langle$ **nocon**, x, y $\rangle$ is sent by NCP B when an $\langle$ **agree**, x, y $\rangle$ message is received and no conversation is open. Assume that NCP A does not receive an $\langle$ **ack**, x, y $\rangle$ message, even after several retransmissions of $\langle$ **agree**, x, y $\rangle$; only $\langle$ **nocon**, x, y $\rangle$ messages are received. Because it is possible that NCP B crashed before it received $\langle$ **agree**, x, y $\rangle$, NCP A is forced to start a new conversation (by sending $\langle$ **data**, m, x $\rangle$) to prevent loss of m! But it is also

possible that NCP B has already delivered m, and the $\langle$ **ack**, $x, y \rangle$ message was lost, in which case a duplicate is introduced.

It is possible to modify the protocol in such a way that NCP A notifies and closes upon receipt of the $\langle$ **nocon**, $x, y \rangle$ message; this prevents duplicates, but may introduce a loss, which is considered even less desirable.

A five-message conversation and comparison. Belsnes [Bel76] gives a five-message protocol that does not lose information and that introduces duplicates only if an NCP actually crashes. Consequently, this is the best possible protocol, seen in the light of the observation that no reliable communication is possible, earlier in this subsection. Because of the excessive overhead (five messages are exchanged by the NCPs to transmit one information unit), it must be doubted whether the five-message protocol must really be preferred to the much simpler two-message protocol. Indeed, because even the five-message protocol may introduce duplicates (when an NCP crashes), the process level must deal with them somehow. So then the two-message protocol, which may introduce duplicates but can be made free of losses if conversation identifications are added as we did for the three-message protocol, may as well be used.

1.3.3 Research Field

There has been ongoing research in distributed algorithms during the last two decades, and considerable progress towards maturity of the field has been made especially during the nineteen-eighties. In previous sections we have pointed at some technical developments that have stimulated research in distributed algorithms, namely, the design of computer networks (both wide- and local-area) and multiprocessor computers. Originally the research was very much aimed towards application of the algorithms in wide-area networks, but nowadays the field has developed crisp mathematical models, allowing application of the results and methods to wider classes of distributed environments. However, the research maintains tight connections with the engineering developments in communication techniques, because the algorithmical results are often sensitive to changes in the network model. For example, the availability of cheap microprocessors has made it possible to construct systems with many identical processors, which has stimulated the study of "anonymous networks" (see Chapter 9).

There are several journals and annual conferences that specialize in results concerning distributed algorithms and distributed computation. Some other journals and conferences do not specialize solely in this subject but contain

a lot of publications in the area nonetheless. The annual symposium on Principles of Distributed Computing (PoDC) has been organized every year since 1982 up to the time of writing in North America, and its proceedings are published by the Association for Computing Machinery, Inc. International Workshops on Distributed Algorithms (WDAG) were held in Ottawa (1985), Amsterdam (1987), Nice (1989), and since then annually with its proceedings being published by Springer-Verlag in the series *Lecture Notes on Computer Science.* In 1998 the name of this conference has changed to Distributed Computing (DISC). The annual symposia on theory of computing (SToC) and foundations of computer science (FoCS) cover all fundamental areas of computer science, and often carry papers on distributed computing. The proceedings of the SToC meetings are published by the Association for Computing Machinery, Inc., and those of the FoCS meetings by the IEEE. The *Journal of Parallel and Distributed Computing (JPDC)* and *Distributed Computing* publish distributed algorithms regularly, and so does *Information Processing Letters (IPL).*

1.4 Outline of the Book

This book was written with the following three objectives in mind.

(1) To make the reader familiar with techniques that can be used to investigate the properties of a given distributed algorithm, to analyze and solve a problem that arises in the context of distributed systems, or to evaluate the merits of a particular network model.
(2) To provide insight into the inherent possibilities and impossibilities of several system models. The impact of the availability of a global time-frame is studied in Section 3.2 and in Chapters 12 and 15. The impact of the knowledge by processes of their identities is studied in Chapter 9. The impact of the requirement of process termination is studied in Chapter 8. The impact of process failures is studied in Part Three.
(3) To present a collection of recent state-of-the-art distributed algorithms, together with their verification and an analysis of their complexity.

Where a subject cannot be treated in full detail, references to the relevant scientific literature are given. The material collected in the book is divided into three parts: Protocols, Fundamental Algorithms, and Fault Tolerance.

Part One: Protocols. This part deals with the communication protocols used in the implementation of computer communication networks and also introduces the techniques used in later parts.

In Chapter 2 the model that will be used in most of the later chapters is introduced. The model is both general enough to be suitable for the development and verification of algorithms and tight enough for proving impossibility results. It is based on the notion of transition systems, for which proof rules for safety and liveness properties can be given easily. The notion of causality as a partial order on events of a computation is introduced and logical clocks are defined.

In Chapter 3 the problem of message transmission between two nodes is considered. First a family of protocols for the exchange of packets over a single link is presented, and a proof of its correctness, due to Schoone, is given. Also, a protocol due to Fletcher and Watson is treated, the correctness of which relies on the correct use of timers. The treatment of this protocol shows how the verification method can be applied to protocols based on the use of timers.

Chapter 4 considers the problem of routing in computer networks. It first presents some general theory about routing and an algorithm by Toueg for the computation of routing tables. Also treated are the Netchange algorithm of Tajibnapis and a correctness proof for this algorithm given by Lamport. This chapter ends with compact routing algorithms, including interval and prefix routing. These algorithms are called *compact* routing algorithms, because they require only a small amount of storage in each node of the network.

The discussion of protocols for computer networks ends with some strategies for avoiding store-and-forward deadlocks in packet-switched computer networks in Chapter 5. The strategies are based on defining cycle-free directed graphs on the buffers in the nodes of the network, and it is shown how such a graph can be constructed using only a modest amount of buffers in each node.

Part Two: Fundamental Algorithms. This part presents a number of algorithmical "building blocks", which are used as procedures in many distributed applications, and develops theory about the computational power of different network assumptions.

Chapter 6 defines the notion of a "wave algorithm", which is a generalized scheme to visit all nodes of a network. Wave algorithms are used to disseminate information through a network, to synchronize the nodes, or to compute a function that depends on information spread over all the nodes.

As it will turn out in later chapters, many distributed-control problems can be solved by very general algorithmic schemes in which a wave algorithm is used as a component. This chapter also defines the time complexity of distributed algorithms and examines the time and message complexity of a number of distributed depth-first search algorithms.

A fundamental problem in distributed systems is *election*: the selection of a single process that is to play a distinguished role in a subsequent computation. This problem is studied in Chapter 7. First the problem is studied for ring networks, where it is shown that the message complexity of the problem is $\Theta(N \log N)$ messages (on a ring of N processors). The problem is also studied for general networks and some constructions are shown by which election algorithms can be obtained from wave and traversal algorithms. This chapter also discusses the algorithm for spanning tree construction by Gallager *et al.*

A second fundamental problem is that of *termination detection*: the recognition (by the processes themselves) that a distributed computation has completed. This problem is studied in Chapter 8. A lower bound on the complexity of solving this problem is proved, and several algorithms are discussed in detail. The chapter includes some classical algorithms (e.g., the ones by Dijkstra, Feijen, and Van Gasteren and by Dijkstra and Scholten) and again a construction is given for obtaining algorithms for this problem from wave algorithms.

Chapter 9 studies the computational power of systems where processes are not distinguished by unique identities. It was shown by Angluin that in this case many computations cannot be carried out by a deterministic algorithm. The chapter introduces probabilistic algorithms, and we investigate what kind of problems can be solved by these algorithms.

Chapter 10 explains how the processes of a system can compute a global "picture", a snapshot, of the system's state. Such a snapshot is useful for determining properties of the computation, such as whether a deadlock has occurred, or how far the computation has progressed. Snapshots are also used to restart a computations after occurrence of an error.

Chapter 11 studies the effect of the availability of directional knowledge in the network, and also gives some algorithms to compute such knowledge.

In Chapter 12 the effect of the availability of a global time concept will be studied. Several degrees of synchronism will be defined and it will be shown that completely asynchronous systems can simulate fully synchronous ones by fairly trivial algorithms. It is thus seen that assumptions about synchronism do not influence the collection of functions that are computable by a distributed system. It will subsequently be shown, however, that there is an

influence on the communication complexity of many problems: the better the synchronism of the network, the lower the complexity of algorithms for these problems.

Part Three: Fault Tolerance. In practical distributed systems the possibility of failure in a component cannot be ignored, and hence it is important to study how well an algorithm behaves if components fail. This subject will be treated in the last part of the book; a short introduction to the subject is given in Chapter 13.

The fault tolerance of asynchronous systems is studied in Chapter 14. A result by Fischer *et al.* is presented; this shows that deterministic asynchronous algorithms cannot cope with even a very modest type of failure, the crash of a single process. It will also be shown that weaker types of faults can be dealt with, and that some tasks are solvable in spite of a crash failure. Algorithms by Bracha and Toueg will be presented; these show that in contrast, randomized asynchronous systems are able to cope with a reasonably large number of failures. It is thus seen that, as is the case for reliable systems (see Chapter 9), randomized algorithms offer more possibilities than deterministic algorithms.

In Chapter 15 the fault tolerance of synchronous algorithms will be studied. Algorithms by Lamport *et al.* show that deterministic synchronous algorithms can tolerate non-trivial failures. It is thus seen that, unlike in the case of reliable systems (see Chapter 12), synchronous systems offer more possibilities than asynchronous systems. An even larger number of faults can be tolerated if processes are able to "sign" their communication to other processes. Consequently, implementing synchronism in an unreliable system is more complicated than in the reliable case, and the last section of Chapter 15 will be devoted to this problem.

Chapter 16 studies the properties of abstract mechanisms, referred to as *failure detectors*, by which processes can obtain an estimated view of crashed and correct processes. We show implementations of such mechanisms and how they can be used to implement specifications in faulty environments.

A different approach to reliability, namely via self-stabilizing algorithms, is followed in Chapter 17. An algorithm is stabilizing if, regardless of its initial configuration, it converges eventually to its intended behavior. Some theory about stabilizing algorithms will be developed, and a number of stabilizing algorithms will be presented. These algorithms include protocols for several graph algorithms such as the computation of depth-first search trees (as in Section 6.4) and the computation of routing tables (as in Chapter 4). Also, stabilizing algorithms for data transmission (as in Chapter 3) have

been proposed, which may indicate that entire computer networks can be implemented using stabilizing algorithms.

Appendices. Appendix A explains the notation used in this book to represent distributed algorithms. Appendix B provides some background in graph theory and graph terminology. The book ends with a list of references and an index of terms.