4.4 DISTRIBUTED OPERATING SYSTEMS

An operating system is a program that manages the resources of a computer system and provides users with a friendly interface to the system. A distributed operating system extends the concepts of resource management and user friendly interface for shared memory computers a step further, encompassing a distributed computing system consisting of several autonomous computers connected by a communication network.

A distributed operating system appears to its users as a centralized operating system for a single machine, but it runs on multiple-independent computers. An identical copy of the operating system (or a different operating system providing similar services) may run at every computer. On the other hand, some computers in the system that serve a special purpose might run an extended version of the operating system. The key concept is *transparency*. In other words, the use of multiple processors and the accessing of remote data should be invisible (transparent) to the user. The user views the system as a *virtual uniprocessor*, and not as a collection of distinct machines [47]. For instance, a user simply submits a job to the distributed operating system through a computer. The distributed operating system performs distributed execution of the job. The user does not know on what computers the job was executed, on what computers the files needed for execution were stored, or how the communication and synchronization among different computers were carried out.

4.5 ISSUES IN DISTRIBUTED OPERATING SYSTEMS

Some important issues that arise in the design of a distributed operating system include the unavailability of up-to-date global knowledge, naming, scalability, compatibility, process synchronization, resource management, security, and structuring of the operating system. These issues are discussed next.

4.5.1 Global Knowledge

In the case of shared memory computer systems, the up-to-date state of all the processes and resources, in other words, the global (entire) state of the system, is completely and accurately known. Hence, the potentially problematic issues that arise in the design of these systems are well understood and efficient solutions to them exist. In distributed computing systems, these same issues take on new dimensions and their solutions become much more complex for the following reasons. Due to the unavailability of a global memory and a global clock, and due to unpredictable message delays, it is practically impossible for a computer to collect up-to-date information about the global state of the distributed computing system [24]. Therefore, a fundamental problem in the design of a distributed operating system is to determine efficient techniques to implement decentralized system wide control, where a computer does not know the current and complete status of the global state. Another significant problem, given the absence of a global clock, is the question of how to order all the events that occur at different times at different computers present in the system. Note that the temporal ordering of events is a fundamental concept in the design and development of distributed systems (e.g., an operating system may schedule jobs based on their time of arrival).

Chapter 5 describes two logical clock schemes which allow the ordering of events in distributed systems despite the absence of a global clock. It also presents a mechanism employed to obtain a global state in distributed systems.

Chapter 8 describes several algorithms to achieve consensus in distributed systems in the absence of global knowledge. For example, a token (a special control message) controls access to shared data in many mutual exclusion algorithms (see Chap. 6), and a token can also be used to control access to the communication network (see Sec. 4.6.2). If the token is lost due to a communication or computer failure, then only one computer should generate a new token and only one token should be generated. This requires that all the computers in the system arrive at the consensus that the token is indeed lost and consequently decide which computer should generate a new token. Note that due to unpredictable communication delays, a token may be in transit, and this may only appear to be lost. Thus, in the absence of global knowledge about the state of the computers and their communication links, arriving at a consensus in distributed systems is a significant challenge.

4.5.2 Naming

Names are used to refer to objects. Objects that can be named in computer systems include computers, printers, services, files, and users. An example of a service is a *name service*. A name service maps a logical name into a physical address by making use of a table lookup, an algorithm, or through a combination of the two [11]. In the implementation of a table lookup, tables (also known as directories) that store names and their physical addresses are used for mapping names to their addresses. In distributed systems, the directories may be replicated and stored at many different locations to overcome a single point of failure as well as to increase the availability of the name service. The two main drawbacks of replication are: (1) It requires more storage capacity, and (2) synchronization requirements need to be met when directories are updated, as the directory at each location would need an updated copy. On the other hand, directories may be partitioned to overcome the drawbacks of replicated directories. The problem with partitioned directories is the difficulty encountered when attempting to find the partition containing a name and address of interest. This may be handled through yet another directory or through a broadcast search [11]. Note, however, that a partitioned directory is less reliable than a replicated directory.

If an algorithm is used for mapping, the algorithm would depend upon the structure of the names. Several examples of name resolving algorithms can be found in Sec. 9.4.1.

Another issue in naming is the method of naming objects such that an object can be located irrespective of its logical name. This topic is treated in Sec. 9.4.1.

4.5.3 Scalability

Systems generally grow with time. The techniques used in designing a system should not result in system unavailability or degraded performance when growth occurs. For example, broadcast based protocols work well for small systems (systems having a small number of computers) but not for large systems. Consider a distributed file system that locates files by broadcasting queries. Under this file system, every computer in the distributed system is subjected to message handling overhead, irrespective of whether it has the requested file or not. As the number of users increase and the system gets larger, the number of file location queries will increase and the overhead will grow larger as well, hurting the performance of every computer. In general, any design approach in which the requirement of a scarce resource (such as storage, communication bandwidth, and manpower) increases linearly with the number of computers in the system, is likely to be too costly to implement. For example, a design requiring that information regarding the system's configuration or directories be stored at every computer is not suitable, even for systems of moderate size [11].

4.5.4 Compatibility

Compatibility refers to the notion of interoperability among the resources in a system. The three different levels of compatibility that exist in distributed systems are the *binary level*, the *execution level*, and the *protocol level* [11].

In a system that is compatible at the binary level, all processors execute the same binary instruction repertoire, even though the processors may differ in performance and in input-output. The Emerald distributed system [21] exhibits binary level compatibility. A significant advantage of binary level compatibility is that it is easier for system development, as the code for many functions provided by the system programs directly depend on the underlying machine level instructions. On the other hand, the distributed system cannot include computers with different architectures from the same or different vendors. Because of this major restriction, binary compatibility is rarely supported in large distributed systems.

Execution level compatibility is said to exist in a distributed system if the same source code can be compiled and executed properly on any computer in the system [11]. Both Andrew [32] and Athena [11] systems support execution level compatibility.

Protocol level compatibility is the least restrictive form of compatibility. It achieves interoperability by requiring all system components to support a common set of protocols. A significant advantage of protocol level compatibility is that individual computers can run different operating systems while not sacrificing their interoperability. For example, a distributed system supporting protocol level compatibility employs common protocols for essential system services such as file access (for example see Sun NFS, Sec. 9.5.1), naming, and authentication.

4.5.5 Process Synchronization

The synchronization of processes in distributed systems is difficult because of the unavailability of shared memory. A distributed operating system has to synchronize processes running at different computers when they try to concurrently access a shared resource, such as a file directory. For correctness, it is necessary that the shared resource be accessed by a single process at a time. This problem is known as the *mutual exclusion* problem, wherein concurrent access to a shared resource by several uncoordinated user requests must be serialized to secure the integrity of the shared resource

Chapter 6 describes several algorithms for achieving mutual exclusion and compares their performance.

In distributed systems, processes can request resources (local or remote) and release resources in any order that may not be known a priori. If the sequence of the allocation of resources to processes is not controlled in such environments, deadlocks may occur. It is important that deadlocks are detected and resolved as soon as possible, otherwise, system performance can degrade severely. Chapter 7 discusses several deadlock handling strategies and describes several deadlock detection techniques for distributed systems.

4.5.6 Resource Management

Resource management in distributed operating systems is concerned with making both local and remote resources available to users in an effective manner. Users of a distributed operating system should be able to access remote resources as easily as they can access local resources. In other words, the specific location of resources should be hidden from the users. The resources of a distributed system are made available to users in the following ways: data migration, computation migration, and distributed scheduling [41].

DATA MIGRATION. In the process of data migration, data is brought to the location of the computation that needs access to it by the distributed operating system. The data in question may be a file (stored locally or remotely) or contents of a physical memory (local or of another computer). If a computation updates a set of data, the original location (remote or local) may have to be similarly updated.

If the data accessed is a file, then the computation's data access request is brought under the purview of the *distributed file system* by the distributed operating system. A distributed file system is the component of a distributed operating system that implements a common file system available to the autonomous computers in the system. The primary goal of a distributed file system is to provide the same functional capability to access files regardless of their location in the network as that provided by a file system of a time-sharing mainframe operating system that only accesses files residing at one location. Ideally, the user doesn't need to be aware of the location of files to access them. This property of a distributed file system is known as *network transparency*. Important issues in the design of a distributed file system, common mechanisms employed in the building of distributed file systems, and several case studies of distributed file systems are presented in Chap. 9.

If, on the other hand, the data accessed is in the physical memory of another computer, then a computation's data access request is brought under the purview of *distributed shared memory* management by the distributed operating system. A distributed shared memory provides a virtual address space that is shared among all the computers in a distributed system. A distributed shared memory is an implementation of the shared memory concept in distributed systems that have no physically shared memory. The major issues in distributed shared memory implementation concern the maintenance of consistency of the shared data and the minimization of delays in the access of data. Distributed shared memory management is discussed in Chap. 10.

COMPUTATION MIGRATION. In computation migration, the computation migrates to another location. Migrating computation may be efficient under certain circumstances. For example, when information is needed concerning a remote file directory, it is more efficient to send a message (i.e., a computation) requesting the necessary information and receive the information back, rather than having the entire directory transferred and then finding the necessary information locally. In distributed scheduling (discussed next), one computer may require another computer's status (such as its load level). It is more efficient and safe to find this information at the remote computer and send the required information back, rather than to transfer the private data structure of the operating system at the remote computer to the requesting computer so that it can obtain the necessary information. The remote procedure call (RPC) mechanism has been widely used for computation migration and for providing communication between computers. The RPC mechanism is discussed in Sec. 4.7.2. Note that in computation migration, only a part of the computation of a process is normally carried out on a different machine.

DISTRIBUTED SCHEDULING. In distributed scheduling, processes are transferred from one computer to another by the distributed operating system. That is, a process may be executed at a computer different from where it originated. Process relocation may be desirable if the computer where a process originated is overloaded or it does not have the necessary resources (such as a math co-processor) required by the process. Distributed scheduling is responsible for judiciously and transparently distributing processes amongst computers such that overall performance is maximized. Improved performance is mainly due to the enhanced utilization of computers through the concurrent execution of processes. Various issues in distributed scheduling, several distributed scheduling algorithms, and case studies of several implementations are discussed in Chap. 11.

4.5.7 Security

The security of a system is the responsibility of its operating system. Two issues that must be considered in the design of security for computer systems are *authentication* and *authorization* [11]. Authentication is the process of guaranteeing that an entity is what it claims to be. Authorization is the process of deciding what privileges an entity has and making only these privileges available. The security of computer systems and various protection mechanisms to achieve security are discussed in Chaps. 14 and 15.

4.5.8 Structuring

The structure of an operating system defines how various parts of the operating system are organized.

THE MONOLITHIC KERNEL. The traditional method of structuring operating systems is to construct them as one big monolithic kernel. This kernel would consist of all the services provided by the operating system. However, in the case of distributed systems that often consist of diskless workstations, workstations with local disk storage, multiprocessor computers suitable for intensive numerical computations, etc., it seems wasteful for every computer to run a huge monolithic operating system when not every computer would use every service provided by the operating system. For example, a diskless workstation would not make use of the storage related operations provided by the file system. This concern has led to the development of the collective kernel structure.

THE COLLECTIVE KERNEL STRUCTURE. In the collective kernel structure, an operating system is structured as a collection of processes that are largely independent of each other [52].

In collective kernel structuring, the operating system services (e.g., distributed memory management, distributed file systems, distributed scheduling, name services, the RPC facility, time management, etc.) are implemented as independent processes. The nucleus of the operating system, also referred to as the *microkernel*, supports the interaction (through messages) between the processes providing the system services. In addition, the microkernel provides services that are typically essential to every computer in a distributed system, such as task management (e.g., local scheduling of tasks), processor management, virtual memory management, etc. The microkernel runs on all the computers in a distributed system. The other processes (all or a few) may or may not run at a computer depending on the need and the hardware available at that computer.

The collective kernel structure can readily make use of a very helpful design technique known as *policy and mechanism separation* [52]. By separating policies and mechanisms in an implementation, one can change any given policy without changing the underlying mechanisms.

Mach [1], V-kernel [12], Chorus [39], and Galaxy [43] are examples of operating systems that use the collective kernel structuring technique.

OBJECT ORIENTED OPERATING SYSTEM. While the various services provided by an operating system can be realized as a set of processes (this model has been referred to as process-model by Goscinski [19]), another popular approach is to implement the services as objects. An operating system that is structured using objects in this manner is known as an *object-oriented operating system.*

In an object-oriented operating system, the system services are implemented as a collection of objects. Each object encapsulates a data structure and defines a set of operations on that data structure. Each object is given a type that designates the properties of the object: process, directory, file, etc. By performing operations defined on an object, the data encapsulated can be accessed and modified. This model is amenable to the collective structuring and policy and mechanism separation techniques. Examples of object oriented operating systems are Eden [3], Choices [10], x-kernel [20], Medusa [34], Clouds [38], Amoeba [48], and Muse [52].

4.5.9 Client-Server Computing Model

In the client-server model, processes are categorized as servers and clients. Servers are the processes that provide services. Processes that need services are referred to as clients. In the client-server model, a client process needing a service (e.g., reading data from a file) sends a message to the server and waits for a reply message. The server process, after performing the requested task, sends the result in the form of a reply message to the client process. Note that servers merely respond to the requests of the clients, and do not typically initiate conversations with clients. In systems with multiple servers, it is desirable that when providing services to clients, the locations and conversations among the servers are transparent to the clients. Clients typically make use of a cache to minimize the frequency of sending data requests to the servers. Systems structured on the client-server model can easily adapt the collective kernel structuring technique.

4.6 COMMUNICATION NETWORKS

This Section introduces the communication aspects of distributed systems. All the computers in a distributed system are interconnected through a computer communication network. A computer can exchange messages with other computers and access data stored at another computer through this network. Communication networks are broadly classified as Wide Area Networks and Local Area Networks.

4.6.1 Wide Area Networks

Wide area networks (WANs) are employed to interconnect various devices (such as computers and terminals) spread over a wide geographic area that may cover different cities, states, and countries. WANs have also been referred to as Long Haul Networks. The communication facility in a WAN consists of switches that are interconnected by communication links. (See Fig. 4.2.) These links may be established through

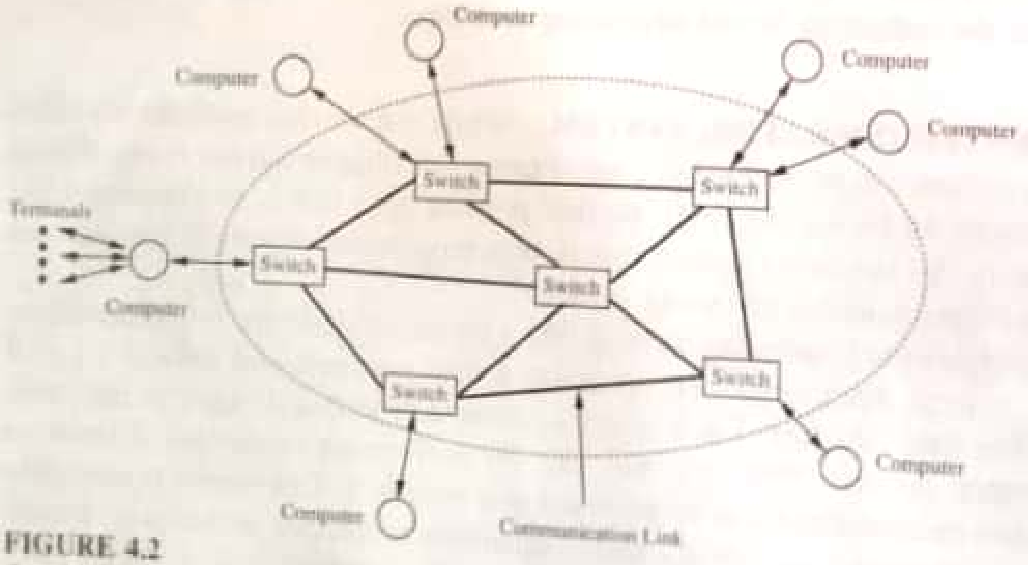

FIGURE 4.2
A point-to-point network.

telephone lines, satellites, microwave links, or any combination of the three. Most WANs employ a technique known as *point-to-point* or *store-and-forward* where data is transferred between computers through a series of switches.

Switches are special purpose computers primarily responsible for routing data from one point to another through an appropriate path while avoiding network congestion. Note that a path or a portion of a path may become congested due to heavy data communication through that path, and/or to limited bandwidth (The bandwidth of a communication link refers to the amount of data that can be communicated over the link in a given amount of time). The data being communicated in a WAN can be lost for any of the following reasons: switch crashes, communication link failure, limited buffer capacity at switches, transmission error, etc.

PACKET SWITCHING VERSUS CIRCUIT SWITCHING. The communication network can be utilized in one of the following two modes, namely, *circuit switching* or *packet switching* [46]. In circuit switching, a dedicated path is established between two devices wishing to communicate, and the path remains intact for the entire duration in which the two devices communicate. The telephone system uses circuit switching. When one subscriber dials another subscriber's number or when one connects his terminal to a computer by dialing the computer's number, a dedicated path between the two points is established through various switches. The path is broken when one side terminates the conversation.

In packet switching, a connection is established between the source device (terminal or computer) and its nearest switch [46]. The data or message to be communicated is broken down into smaller units called packets (several hundred to several thousands bytes in length), with each packet containing the address of the destination. The packets are then sent to the nearest switch. These packets are routed from one switch to another switch in the communication network until they arrive at the switch connected to the destination device, at which point the data is delivered to the destination. Thus, in packet switching, a communication path (switches and links) is not reserved by any two devices wishing to communicate, but rather is dynamically shared among many devices on a demand basis. The achievable utilization of the communication network is higher under packet switching compared to circuit switching because the network can be shared by many devices. Also, parallel transmission, and hence reduction in data transmission time, is possible because packets forming one message may travel along different paths. Moreover, data transmission in computer networks is sporadic rather than continuous. Thus, most computer networks use packet switching to permit better utilization of the network. One disadvantage of packet switching, however, is that the breaking of a message into packets and assembling them back at the destination carries some cost.

THE ISO OSI REFERENCE MODEL. Generally, WANs must interconnect heterogeneous types of equipment (e.g., computers, terminals, printers). These types of equipment may differ from each other in their speed, word length, information representation, or in many other criteria. To communicate in such a heterogeneous environment, the

ISO OSI reference model[†] provides a framework for communication protocols [53]. It organizes the protocols as seven layers and specifies the functions of each layer. The organization of these seven layers is shown in Fig. 4.3. In this model, user programs run in the application layer.

When an application program running at computer A wants to send a message to an application program at computer B, the following chain of events occur. The application at computer A passes the message down to the presentation layer (on computer A itself). The presentation layer transforms the data (explained later), adds a header containing some control information to the message, and passes the resulting message to the session layer. The session layer adds its owner header to the message and passes the resulting message to the next layer. This continues on until the message reaches the physical layer. The physical layer on computer A transmits the raw data bits to the physical layer running at computer B. Note that the message is routed to compute B through various intermediate switches in the communication network. Once the message is received at computer B, the protocol at each layer strips the header added by its counterpart at computer A, performs the necessary processing identified by the header, and passes the message on to the next layer. This continues until the message reaches its destination—a process in the application layer at computer B.

The ISO OSI model does not specify how the layers should be implemented. Every layer is aware only of the protocols and header formats of its counterpart. It does not understand the header or the protocols used by the other layers. This makes each layer independent, so any layer can change its protocol without affecting other layers as long as the interfaces between the layers remain unchanged.

We next give a brief overview of the ISO OSI model [46]. Complete information about the model can be found in [46, 53].

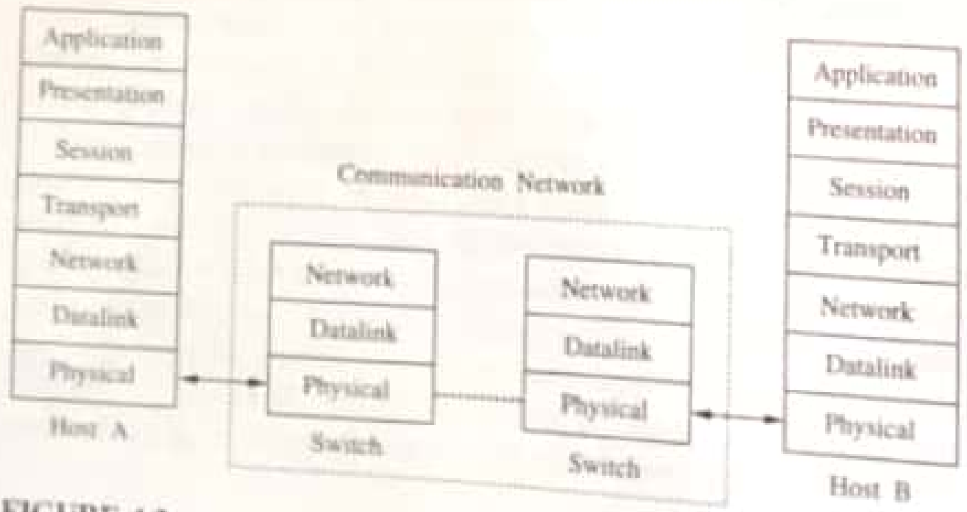

FIGURE 4.3
The ISO OSI reference model.

[†]International Standards Organization's Reference Model of Open Systems Interconnection.

Session:- temporary and ~~[illegible]~~ interactive info b/w two or more communicating devices, or b/w a computer and user eg login session

AN OVERVIEW OF THE ISO OSI LAYERS. The *physical* layer's function is to allow a device to send raw bit streams of data over the communication network. It is not concerned with transmission errors, how bits are organized, or what they mean. This layer should, however, be aware of and take care of the communication network implementation details such as circuit/packet switching, the type of network (i.e., telephone system, digital transmission, etc.), the voltage levels used for representing 0 and 1 bits, the number of pins and their assignments in network connectors, etc.

The *data-link* layer is responsible for recovering from transmission errors and for flow control. Flow control takes care of any disparity between the speeds at which the bits can be sent and received. The data-link layer makes the communication facility provided by the physical layer reliable.

The *network* layer is mainly responsible for routing and congestion control. It breaks a message into packets and decides which outgoing line will carry the packets toward their destination.

The *transport* layer's primary function is to hide all the details of the communication network from the layers above. It provides a network independent device-to-device (or end-to-end) communication. This layer can provide the ability to the network to inform a host that the network has crashed or has lost certain packets. Thus, the transport layer can provide improved reliability if necessary.

The *session* layer is responsible for establishing and maintaining a connection, known as a session, between two processes. Establishing a connection may involve the authentication of the communicating processes and the selection of the right transport service. In addition, the session layer may keep track of the outstanding requests and replies from processes and order them in such a manner to simplify the design of user programs.

The *presentation* layer is the interface between a user program and the rest of the network. It provides data transformation utilities to take care of the differences in representing information at the source and at the destination. In addition, the presentation layer may perform data compression, encryption, and conversion to and from network standards for terminals and files.

The *application* layer's function is to provide a facility for the user processes to use the ISO OSI protocols. Its content is left to the users. (For example, specific applications such as airline and banking may have their own standards for the application layer.)

4.6.2 Local Area Networks

A local area network (LAN) is a communication network that interconnects a variety of data communication devices within a small geographic area [45]. In our context, the data communicating devices typically include computers, terminals, and peripheral devices. Some of the key characteristics of LANs are:

- High data transmission rates (10 megabits per second to 100 megabits per second).
- The geographic scope of LANs is small, generally confined to a single building or perhaps several buildings (such as a college campus).
- Low transmission error rate.

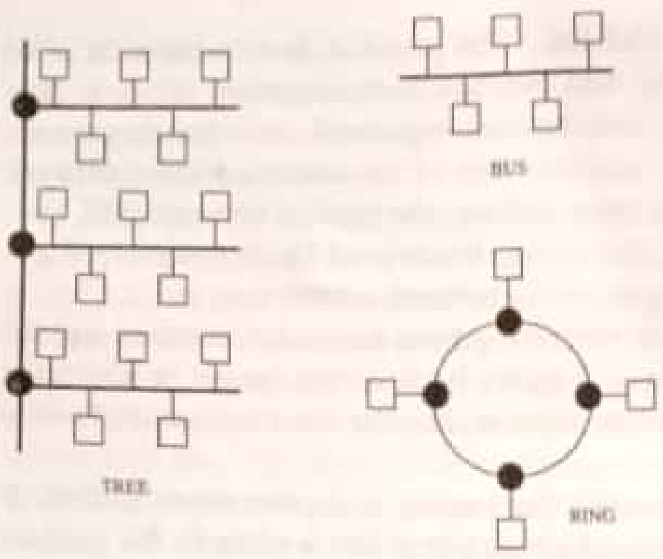

FIGURE 4.4
Network topologies.

Widely used network topologies for LANs are bus, ring, and tree (See Fig. 4.4). The communication media can be coaxial cable, twisted pair wire, or optical fiber.

BUS/TREE TOPOLOGY. In the bus topology, the communication devices transmit data in the form of packets where each packet contains the address of the destination and a message. A packet propagates throughout the medium, the bus, and is available to all the other devices, but is received only by the addressed device [45]. A tree topology LAN is obtained by interconnecting many bus topology LANs to a common bus (see Fig. 4.4). Bus topology LANs can be viewed as branches of a tree. In the bus topology, all the devices connected to the LAN are allowed to transmit at any time. However, as the devices share a common data path (i.e., the bus), a protocol to control the access to the bus is necessary. We next discuss two protocols to control access to the bus.

The CSMA/CD protocol. The most commonly used *access control protocol* for bus topology is CSMA/CD (Carrier Sense Multiple Access with Collision Detection) [45]. Under this protocol, a device wishing to transmit listens to the medium to determine whether another transmission is in progress. If so, the device waits for a random amount of time before trying again. If no other transmission is in progress, the device starts transmitting data and continues to listen to the medium while it is transmitting. If another device starts transmitting simultaneously, the two transmissions collide. If a collision is detected, a short jamming signal is transmitted over the bus to inform all the devices that there has been a collision. The devices will then wait for a random amount of time before attempting to transmit again. The principal advantage of this protocol is its simplicity. Its principal disadvantage is that under a heavy load, contention for the bus rises and performance degrades because of frequent collisions. Thus, a bus using the CSMA/CD protocol cannot support a large number of devices per bus. Ethernet is an example of a LAN that is based on the CSMA/CD principle [31].

When the CSMA/CD protocol is used for a tree topology, packets transmitted by one device to another will not enter the common bus unless the destination device is on another branch of the tree. Hence, the common bus serves as a backbone connecting many LANs. In large systems, the backbone is normally a high speed medium with a bandwidth of 100 megabits per second.

The token bus protocol. An alternative to the CSMA/CD protocol is the token bus technique [45]. In this technique, devices physically organized in a bus/tree topology form a *logical* ring, and each device knows the identity of the devices preceding and following it on the ring. Access to the bus is controlled through a token (a control packet). The device holding the token is allowed to transmit, poll other devices, and receive replies from other devices. The devices not holding the token can receive messages and can only respond to polls or requests for acknowledgment. A device is allowed to keep the token for a specific amount of duration, after which it has to send the token to the device following it on the logical ring.

RING TOPOLOGY. The main alternative to the bus/tree topology is the ring topology (see Fig. 4.4). Note that, in the token bus protocol, the ring is logical, whereas in the ring topology, the ring is physical. In this topology, data is transmitted point-to-point. At each point, the address on the packet is copied and checked to see if the packet is meant for the device connected at that point. If the address of the device and the address in the packet match, the rest of the packet is copied, otherwise, the entire packet is retransmitted to the next device on the ring.

The token ring protocol. A widely used access control protocol to control access to the ring is the token ring technique [17, 45]. Under this technique, a token circulates around the ring. The token is labeled *free* when no device is transmitting. When a device wishes to transmit, it waits for the token to arrive, labels the token as *busy* on arrival, and retransmits the token. Immediately following the release of the token, the device transmits data. The transmitting device will mark the token as *free* when the busy token returns to the device and the device has completed its transmission. The main advantage of the token ring protocol is that it is not sensitive to the load on the network; the entire bandwidth of the medium can be utilized. The major disadvantage of the token ring protocol is its complexity. The token has to be maintained error-free. If the token is lost, care must be taken to generate only one token. The maintenance of the token may require a separate process to monitor it.

The slotted ring protocol. The slotted ring is another technique used to control access to a ring network [37, 45]. In this technique, a number of fixed length slots continuously circulate around the ring. The ring is like a conveyor belt. A device wishing to transmit data waits for a slot marked *empty* to arrive, marks it *full*, and inserts the destination's address and the data into the slot as it goes by. The device is not allowed to retransmit again until this slot returns to the device, at which time it is marked as empty by the device. As each device knows how many slots are circulating around the ring, it can determine the slot it had marked previously. After the newly emptied slot continues on, the device is again free to transmit data. A few bits are reserved in each slot so that the result of the transmission (accepted, busy, or rejected) can be returned to the source.

The key advantage of the slotted ring technique is its simplicity, which translates into reliability. The prime disadvantage is wasted bandwidth. When the ring is not heavily utilized, many empty slots will be circulating, but a particular device wishing to transmit considerable amounts of data can only transmit once per round-trip ring time.

4.7 COMMUNICATION PRIMITIVES

The communication network provides a means to send raw bit streams of data in distributed systems. The communication primitives are the high-level constructs with which programs use the underlying communication network. They play a significant role in the effective usage of distributed systems. The communication primitives influence a programmer's choice of algorithms as well as the ultimate performance of the programs. They influence both the ease of use of a system and the efficiency of applications that are developed for the system [29].

We next discuss two communication models, namely, message passing and remote procedure call, that provide communication primitives. These two models have been widely used to develop distributed operating systems and applications for distributed systems.

4.7.1 The Message Passing Model

The message passing model provides two basic communication primitives, namely, SEND and RECEIVE [47]. The SEND primitive has two parameters, a message and its destination. The RECEIVE primitive has two parameters, the source (including anyone) of a message and a buffer for storing the message. An application of these primitives can be found in the client-server computation model. In the client-server model, a client process needing some service (e.g., reading data from a file) sends a message to the server and waits for a reply message. After performing the task, the server process sends the result in the form of a reply message to the client process. While these two primitives provide the basic communication ability to programs, the semantics of these primitives also play a significant role in ease of developing programs that use them. We next discuss two design issues that decide the semantics of these two primitives.

BLOCKING VS. NONBLOCKING PRIMITIVES. In the standard message passing model, messages are copied three times: from the user buffer to the kernel buffer, from the kernel buffer on the sending computer to the kernel buffer on the receiving computer, and finally from the buffer on the receiving computer to a user buffer [29]. This is known as the *buffered* option.

With *nonblocking* primitives, the SEND primitive returns control to the user process as soon as the message is copied from the user buffer onto the kernel buffer. The corresponding RECEIVE primitive signals its intention to receive a message and provides a buffer to copy the message. The receiving process may either periodically check for the arrival of a message or be signaled by the kernel upon arrival of a message. The primary advantage of nonblocking primitives is that programs have maximum flexibility to perform computation and communication in any order they want. On the other hand, a significant disadvantage of nonblocking primitives is that programming becomes tricky and difficult. Programs may become time-dependent where problems (or system states) are irreproducible, making the programs very difficult to debug [47].

In the *unbuffered* option, data is copied from one user buffer to another user buffer directly [29]. In this case, a program issuing a SEND should avoid reusing the user buffer until the message has been transmitted. For large messages (thousands of bytes), a combination of unbuffered and nonblocking semantics allows almost complete overlap between the communication and the ongoing computational activity in the user program.

A natural use of nonblocking communication occurs in producer-consumer relationships. The consumer process can issue a nonblocking RECEIVE. If a message is present, the consumer process reads it, otherwise it performs some other computation. The producer process can issue nonblocking SENDs. If a SEND fails for any reason (e.g., the buffer is full), it can be retried later.

With *blocking* primitives, the SEND primitive does not return control to the user program until the message has been sent (an *unreliable* blocking primitive) or until an acknowledgment has been received (a *reliable* blocking primitive). In both cases, the user buffer can be reused as soon as the control is returned to the user program. The corresponding RECEIVE primitive does not return control until a message is copied to the user buffer. A reliable RECEIVE primitive automatically sends an acknowledgment, while an unreliable RECEIVE primitive does not send an acknowledgment [47]. The primary advantage of employing blocking primitives is that the behavior of the programs is predictable, and hence programming is relatively easy. The primary disadvantage is the lack of flexibility in programming and the absence of concurrency between computation and communication.

SYNCHRONOUS VS. ASYNCHRONOUS PRIMITIVES. We now address the question of whether to buffer or not to buffer messages. With *synchronous* primitives, a SEND primitive is blocked until a corresponding RECEIVE primitive is executed at the receiving computer. This strategy is also referred to as a *rendezvous*. A blocking-synchronous primitive can be extended to an unblocking-synchronous primitive by first copying the message to a buffer at the sending side, and then allowing the process to perform other computational activity except another SEND.

With *asynchronous* primitives, the messages are buffered. A SEND primitive does not block even if there is no corresponding execution of a RECEIVE primitive. The corresponding RECEIVE primitive can either be a blocking or a nonblocking primitive. One disadvantage of buffering messages is that it is more complex, as it involves creating, managing, and destroying buffers. Another issue is what to do with the messages that are meant for processes that have already died [47].

4.7.2 Remote Procedure Calls

While the message passing communication model provides a highly flexible communication ability, programmers using such a model must handle the following details.

- Pairing of responses with request messages.
- Data representation (when computers of different architectures or programs written in different programming languages are communicating).

- Knowing the address of the remote machine or the server.
- Taking care of communication and system failures.

The handling of all these details in programs makes the development of programs for distributed computations difficult. In addition, these programs can be time-dependent, making it almost impossible to reproduce errors and debug. These difficulties led to the development of the remote procedure call (RPC) mechanism [9]. RPC mechanisms hide all the above details from programmers. The RPC mechanism is based on the observation that procedure call is a well known and well understood mechanism for transfer of control and data within a program running on a single computer (a shared memory system). The RPC mechanism extends this same mechanism to transfer control and data across a communication network (a non-shared memory system) [9]. A remote procedure call can be viewed as an interaction between a client and a server, where the client needing a service invokes a procedure at the server. A simple RPC mechanism works as follows.

Basic RPC operation. On invoking a remote procedure, the calling process (the client) is suspended and parameters, if any, are passed to the remote machine (the server) where the procedure will execute. On completion of the procedure execution, the results are passed back from the server to the client and the client resumes execution as if it had called a local procedure. While the RPC mechanism looks simple, the issues that arise in designing and implementing it are not so simple. We next discuss several of those issues.

4.7.3 Design Issues in RPC

Structure. A widely used organization for RPC mechanisms is based on the concept of *stub* procedures [4, 9, 40] (see Fig. 4.5). When a program (client) makes a remote procedure call, say *P(x,y)*, it actually makes a local call on a dummy procedure or a client-stub procedure corresponding to procedure *P*. The client-stub procedure constructs a message containing the identity of the remote procedure and parameters, if any, to be passed. It then sends the message to the remote server machine (a primitive similar to SEND explained in Sec. 4.7.1 may be used for this purpose). A stub procedure at the remote machine receives the message (a primitive similar to RECEIVE may be used) and makes a local call to the procedure specified in the message and passes the parameters received to the procedure. When the remote procedure completes execution, the control returns to the server-stub procedure. The server-stub procedure passes the results back to the client-stub procedure at the calling machine, which returns the results to the client. The stub procedures can be generated at compile time or can be linked at run time.

Binding. Binding is a process that determines the remote procedure, and the machine on which it will be executed, upon a remote procedure invocation. The binding process may also check the compatibility of the parameters passed and the procedure type called with what is expected from the remote procedure.

One approach for binding in the client-server model makes use of a binding server [9, 44] (see Fig. 4.5). The servers register the services they provide with the binding server. The binding server essentially stores the server machine addresses along with the services they provide. When a client makes a remote procedure call, the client-stub procedure obtains the address of the server machine by querying the binding server. Another approach used for binding is where the client specifies the machine and the service required, and the binding server returns the port number required for communicating with the service requested [4]. Note that the first method is location transparent while the latter is not.

Parameter and result passing. To pass parameters or results to a remote procedure, a stub procedure has to convert the parameters and results into an appropriate representation (a representation that is understood by the remote machine) first and then pack them into a buffer in a form suitable for transmission. After the message is received, the message must be unpacked (see Fig. 4.5).

Converting data into an appropriate representation becomes expensive if it has to be done on every call. One way to avoid conversions is to send the parameters along with a code identifying the format used so that the receiver can do the conversion, (only, of course, if it uses a different representation). This approach requires the machine to know how to convert all the formats that can possibly be used. This approach also has poor portability because whenever a new representation (because of a new machine type or a new language) is introduced into the system, existing software needs to be updated [47].

Alternatively, each data type may have a standard format in the message. In this technique, the sender will convert the data to the standard format and the receiver will convert from the standard format to its local representation. With this approach a machine doesn't need to know how to convert all the formats that can possibly be used.

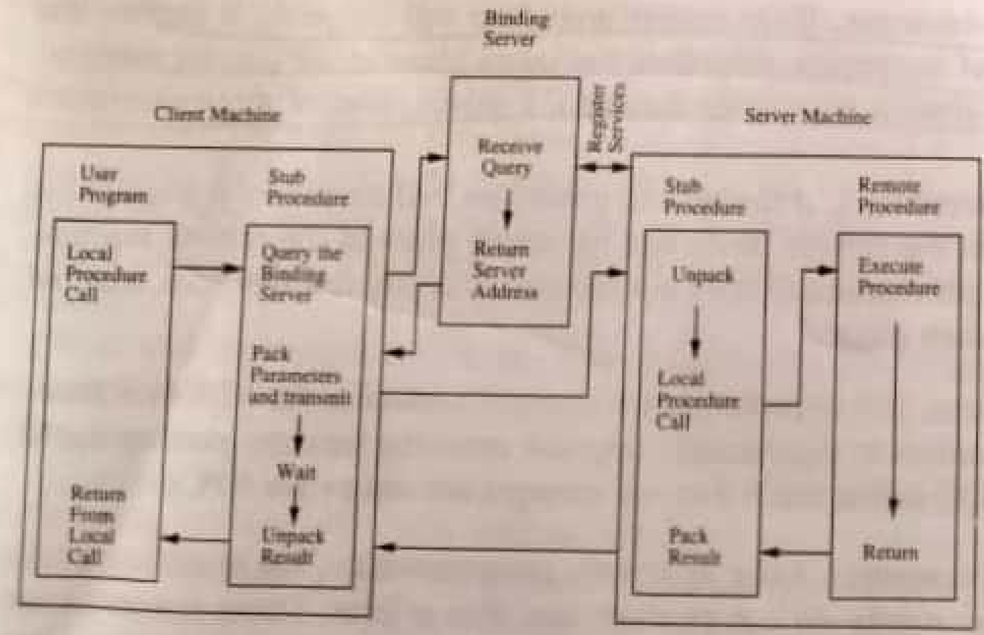

FIGURE 4.5
Remote procedure call.

This method, however, is wasteful if both the sender and the receiver use the same internal representation [47].

Another issue is how to deal with passing parameters by value and by reference. Passing parameters by value is simple, as the stub procedure simply copies the parameters into the message. However, passing parameters by reference is exceedingly more complicated. For example, just passing file pointers is inadequate, because the privileges associated with the calling process also have to be passed [47]. Moreover, the semantics associated with passing parameters in local procedure calls and remote procedure calls can be different. For example, in Argus [28], the structures pointed to by pointers are copied onto the remote machine, and hence, the calling machine and the remote machine do not share the data structures directly.

Error handling, semantics, and correctness. A remote procedure call can fail for at least two reasons: computer failures and communication failures (such as when a transmitted message does not reach its destination).

Handling failures in distributed systems is difficult. For example, consider the case where messages are lost occasionally. If the remote server is slow, the program that invokes the remote procedure may invoke the remote procedure more than once, suspecting a message loss. This could result in more than one execution of the procedure at the remote machine. Also, consider the case where the client machine, after having invoked a remote procedure, crashes immediately. In this case, the remote procedure is executed in vain, as there is no machine to receive the result. On the other hand, if the client machine recovers quickly and reissues the remote procedure call, then there is a possibility of more than one execution of the remote procedure. The unwanted executions have been referred to as *orphans* [22].

In view of the above problems associated with distributed system failure, it is clear that the semantics of RPCs play a significant role in the ease of development of programs for distributed computation. The semantics of RPCs are classified as follows [33, 36]:

"At least once" semantics. If the remote procedure call succeeds, it implies that at least one execution of the remote procedure has taken place at the remote machine. If the call does not succeed, it is possible that zero, a partial, one, or more executions have taken place.

"Exactly once" semantics. If the remote procedure call succeeds, it implies that exactly one execution of the remote procedure has taken place at the remote machine. However, if the remote procedure call does not succeed, it is possible that zero, a partial, or one execution has taken place.

In view of the above, it is apparent that a stronger semantics for RPCs are necessary for the RPC mechanism to significantly improve upon the message passing model. Liskov and Scheifler [28] define the following stronger semantics for RPCs.

"At most once" semantics. Same as exactly once semantics, but in addition, calls that do not terminate normally do not produce any side effects. These semantics are also referred to as **Zero-or-one** semantics. A number of RPC mechanisms implemented support at-most-once semantics [2, 5, 9, 15, 36, 44].

latency → delay before a transfer of data begins following an instruction for the transfer

Correctness condition. Panzieri and Srivastava [36] define a simple correctness condition for remote procedure calls as follows:

Let C_i denote a call made by a machine and W_i represent the corresponding computation invoked at the called machine.

Let C_2 happen after C_1 (denoted by $C_1 \rightarrow C_2$) and computations W_1 and W_2 share the same data such that W_1 and/or W_2 modify the shared data.

To be correct in the presence of failures, an RPC implementation should satisfy the following *correctness criterion.*

$$C_1 \rightarrow C_2 \text{ implies } W_1 \rightarrow W_2.$$

OTHER ISSUES. The RPC mechanisms make use of the communication facility provided by the underlying network to pass messages to remote machines. One of the issues to be resolved is whether to build the RPC mechanism on top of a flow-controlled and error-controlled virtual-circuit mechanism (similar to establishing sessions in WANs) or directly on top of an unreliable, connectionless (datagram) service [47]. (In a *datagram* service, a machine simply sends a message in the form of packets to the destination, and there is no extensive two-way communication such as automatic acknowledgments.)

As RPC mechanisms became a widely accepted method for communication in distributed systems, a need to specify a remote procedure call as low-latency or high-throughput became necessary, depending on the application. Low-latency calls require minimum round-trip delay and are made in case of infrequent calls (such as calls to a mail-server). On the other hand, the aim of high-throughput calls is to obtain maximum possible throughput from the underlying communication facility. This type of call is typically made when bulk data transfer is required, such as in the case of calls to file servers. The ASTRA RPC mechanism [4] provides the ability to a user to specify whether low-latency or high-throughput is desired. For high-throughput calls, ASTRA makes use of virtual circuit (TCP) protocol. For low-latency calls, ASTRA makes use of a datagram facility that is more suitable for intermittent exchange due to its simplicity. Stream [28] is another RPC mechanism designed mainly to achieve high-throughput. It also makes use of the TCP protocol. In both the ASTRA and Stream implementations, high-throughput is achieved by buffering the messages and immediately returning control to the user. The user can then make more calls. The buffer is flushed when it is full or convenient. By buffering the messages, the overhead to process the communication protocols on every remote procedure call is avoided. For low-latency calls, the buffer is flushed immediately. Future [51] is an RPC facility that is specifically designed for low-latency.

Note that invoking a remote procedure call blocks the calling process. However, these semantics severely limits the concurrency that can be achieved. Several RPC designs have tried to overcome this limitation.

One way to achieve parallelism is through creating multiple processes for each remote procedure call [6]. This scheme allows a process to make multiple calls to many servers and still execute in parallel with the servers. However, creating processes, switching between processes, and destroying processes may not be economical under all circumstances. This approach also does not scale well for large systems consisting of hundreds of computers [40].

In MultiRPC [40], a process is allowed to invoke a procedure on many servers concurrently but not two different procedures in parallel. The calling process is blocked until all the responses are received or until the call is explicitly terminated by the calling process.

Parallel procedure call (PARPC) [30] is another scheme which is similar to MultiRPC. In PARPC, invoking a parallel procedure call executes a procedure in n different address spaces (e.g., at n different servers) in parallel. The caller remains blocked while the n procedures execute. When a result becomes available, the caller is unblocked to execute a statement to process the result of the returned call. After executing the statement, the caller reblocks to wait for the next result. The caller resumes execution after all the n calls have returned or when the caller explicitly terminates the PARPC call.

To overcome the limitations of blocking semantics, asynchronous RPC mechanisms have been developed where a calling process does not block after invoking a remote procedure [4, 44]. ASTRA [4] offers further flexibility by allowing client processes to accept replies in any order. The main disadvantage of these semantics is that, like in message passing primitives, programming becomes difficult.

Bershad et al. [7] performed a study to determine the frequency of intermachine procedure calls. According to their study, less than ten percent of all system calls cross the machine boundary. Note that intramachine calls can be made efficient by avoiding the marshaling of data and other RPC related network protocols. ASTRA optimizes the intramachine call by avoiding the above overhead and by using the most efficient interprocess communication mechanism provided by the host operating system to pass messages.

According to Gifford [18] existing RPC facilities have the following two shortcomings:

- *Incremental results:* In the present RPC facilities, a remote procedure cannot easily return incremental results to the calling process while its execution is still in progress.
- *Protocol flexibility:* In present RPC systems, remote procedures are not *first-class* objects. (A first-class object is a value that can be freely stored in memory, passed as a parameter to both local and remote procedures, and returned as a result from both local and remote procedures [18].) This feature can make protocols inflexible. For example, the following protocol is not implementable unless remote procedures are first-class objects: A process wishes to provide a server with a procedure for use under certain circumstances, and the server then wishes to pass this procedure on to another server.

To overcome the above limitations, Gifford has proposed a new communication model called the *channel model* [18]. In this model, remote procedures are first-class objects. An abstraction called a *pipe* permits the efficient transportation of bulk data and incremental results, and an abstraction called *channel groups* allows the sequencing of calls on pipes and procedures. Complete details on this model are beyond the scope of this book and can be found in [18].

5.3 LAMPORT'S LOGICAL CLOCKS

Lamport [12] proposed the following scheme to order events in a distributed system using logical clocks. The execution of processes is characterized by a sequence of events. Depending on the application, the execution of a procedure could be one event or the execution of an instruction could be one event. When processes exchange messages, sending a message constitutes one event and receiving a message constitutes one event.

Definitions

Due to the absence of perfectly synchronized clocks and global time in distributed systems, the order in which two events occur at two different computers cannot be determined based on the local time at which they occur. However, under certain conditions, it is possible to ascertain the order in which two events occur based solely on the behavior exhibited by the underlying computation. We next define a relation that orders events based on the behavior of the underlying computation.

HAPPENED BEFORE RELATION ($\rightarrow$). The *happened before* relation captures the causal dependencies between events, i.e., whether two events are causally related or not. The relation $\rightarrow$ is defined as follows:

- $a \rightarrow b$, if a and b are events in the same process and a occurred before b.
- $a \rightarrow b$, if a is the event of sending a message m in a process and b is the event of receipt of the same message m by another process.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$, i.e., "$\rightarrow$" relation is transitive.

In distributed systems, processes interact with each other and affect the outcome of events of processes. Being able to ascertain order between events is very important for designing, debugging, and understanding the sequence of execution in distributed computation. In general, an event changes the system state, which in turn influences the occurrence and outcome of future events. That is, past events influence future events and this influence among causally related events (those events that can be ordered by '$\rightarrow$') is referred to as *causal affects*.

CAUSALLY RELATED EVENTS. Event a causally affects event b if $a \rightarrow b$.

CONCURRENT EVENTS. Two distinct events a and b are said to be concurrent (denoted by $a \| b$) if $a \not\rightarrow b$ and $b \not\rightarrow a$. In other words, concurrent events do not causally affect each other.

For any two events a and b in a system, either $a \rightarrow b$, $b \rightarrow a$, or $a \| b$.

Example 5.2. In the space-time diagram of Fig. 5.2, e_{11}, e_{12}, e_{13}, and e_{14} are events in process P_1 and e_{21}, e_{22}, e_{23}, and e_{24} are events in process P_2. The arrows represent message transfers between the processes. For example, arrow $e_{12}e_{23}$ corresponds to a message sent from process P_1 to process P_2, e_{12} is the event of sending the message at P_1, and e_{23} is the event of receiving the same message at P_2. In Fig. 5.2, we see that $e_{22} \rightarrow e_{13}$, $e_{13} \rightarrow e_{14}$, and therefore $e_{22} \rightarrow e_{14}$. In other words, event e_{22} causally

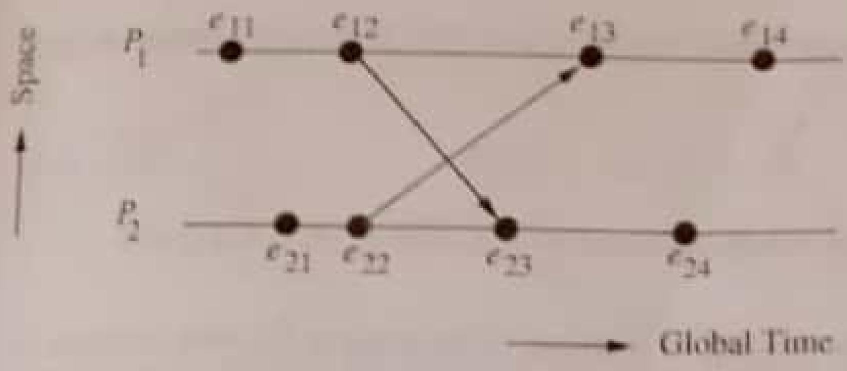

FIGURE 5.2
A space-time diagram.

affects event e_{14}. Note that whenever $a \rightarrow b$ holds for two events a and b, there exists a path from a to b which moves only forward along the time axis in the space-time diagram. Events e_{21} and e_{11} are concurrent even though e_{11} appears to have occurred before e_{21} in real (global) time for a global observer.

Logical Clocks

In order to realize the relation $\rightarrow$, Lamport [12] introduced the following system of logical clocks. There is a clock C_i at each process P_i in the system. The clock C_i can be thought of as a function that assigns a number $C_i(a)$ to any event a, called the *timestamp* of event a, at P_i. The numbers assigned by the system of clocks have no relation to physical time, and hence the name logical clocks. The logical clocks take monotonically increasing values. These clocks can be implemented by counters. Typically, the timestamp of an event is the value of the clock when it occurs.

CONDITIONS SATISFIED BY THE SYSTEM OF CLOCKS. For any events a and b:

$$\text{if } a \rightarrow b, \text{ then } C(a) < C(b)$$

The happened before relation '$\rightarrow$' can now be realized by using the logical clocks if the following two conditions are met:

[C1] For any two events a and b in a process P_i, if a occurs before b, then

$$C_i(a) < C_i(b)$$

[C2] If a is the event of sending a message m in process P_i and b is the event of receiving the same message m at process P_j, then

$$C_i(a) < C_j(b)$$

The following implementation rules (IR) for the clocks guarantee that the clocks satisfy the correctness conditions C1 and C2:

[IR1] Clock C_i is incremented between any two successive events in process P_i:

$$C_i := C_i + d \quad (d > 0) \tag{5.1}$$

If a and b are two successive events in P_i and $a \rightarrow b$, then $C_i(b) = C_i(a) + d$.

[IR2] If event a is the sending of message m by process P_i, then message m is assigned a timestamp $t_m = C_i(a)$ (note that the value of $C_i(a)$ is obtained after applying rule IR1). On receiving the same message m by process P_j, C_j is set to a value greater than or equal to its present value and greater than t_m.

$$C_j := \max(C_j, t_m + d) \quad (d > 0) \tag{5.2}$$

Note that the message receipt event at P_j increments C_j as per rule IR1. The updated value of C_j is used in Eq. 5.2. Usually, d in Eqs. 5.1 and 5.2 has a value of 1.

Lamport's happened before relation, $\rightarrow$, defines an irreflexive partial order among the events. The set of all the events in a distributed computation can be totally ordered (the ordering relation is denoted by $\Rightarrow$) using the above system of clocks as follows: If a is any event at process P_i and b is any event at process P_j then $a \Rightarrow b$ if and only if either

$$C_i(a) < C_j(b) \quad \text{or}$$
$$C_i(a) = C_j(b) \quad \text{and} \quad P_i \prec P_j$$

where $\prec$ is any arbitrary relation that totally orders the processes to break ties. A simple way to implement $\prec$ is to assign unique identification numbers to each process and then $P_i \prec P_j$, if $i < j$.

Lamport's mutual exclusion algorithm, discussed in Sec. 6.6, illustrates the use of the ability to totally order the events in a distributed system.

Example 5.3. Figure 5.3 gives an example of how logical clocks are updated under Lamport's scheme. Both the clock values C_{P_1} and C_{P_2} are assumed to be zero initially and d is assumed to be 1. e_{11} is an internal event in process P_1 which causes C_{P_1} to be incremented to 1 due to IR1. Similarly, e_{21} and e_{22} are two events in P_2 resulting in $C_{P_2} = 2$ due to IR1. e_{16} is a message send event in P_1 which increments C_{P_1} to 6 due to IR1. The message is assigned a timestamp = 6. The event e_{25}, corresponding to the receive event of the above message, increments the clock C_{P_2} to 7 (max(4+1,6+1)) due to rules IR1 and IR2. Similarly, e_{24} is a send event in P_2. The message is assigned a timestamp = 4. The event e_{17} corresponding to the receive event of the above message increments the clock C_{P_1} to 7 (max(6+1, 4+1)) due to rules IR1 and and IR2.

VIRTUAL TIME. Lamport's system of logical clocks implements an approximation to global/physical time, which is referred to as virtual time. Virtual time advances along

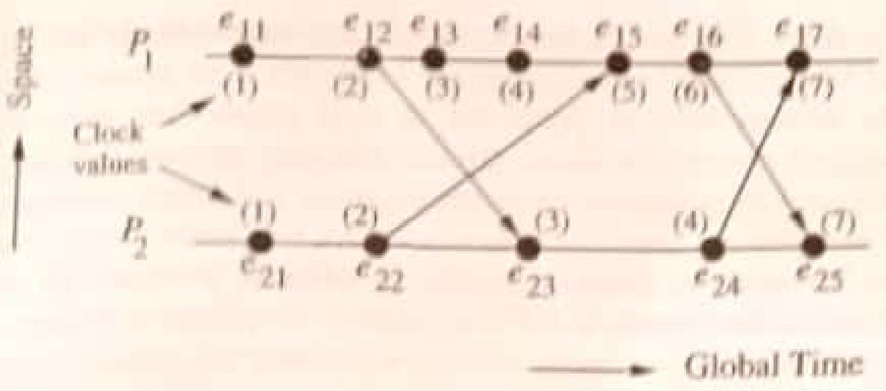
Space
P_1
P_2
e_{11} e_{12} e_{13} e_{14} e_{15} e_{16} e_{17}
(1) (2) (3) (4) (5) (6) (7)
Clock values
(1) (2) (3) (4) (7)
e_{21} e_{22} e_{23} e_{24} e_{25}
Global Time

FIGURE 5.3
How Lamport's logical clocks advance.

with the progression of events and is therefore discrete. If no events occur in the system, virtual time stops, unlike physical time which continuously progresses. Therefore, to wait for a virtual time instant to pass is risky as it may never occur [16].

5.3.1 A Limitation of Lamport's Clocks

Note that in Lamport's system of logical clocks, if $a \rightarrow b$ then $C(a) < C(b)$. However, the reverse is not necessarily true if the events have occurred in different processes. That is, if a and b are events in different processes and $C(a) < C(b)$, then $a \rightarrow b$ is not necessarily true; events a and b may be causally related or may not be causally related. Thus, Lamport's system of clocks is not powerful enough to capture such situations. The next example illustrates this limitation of Lamport's clocks.

Example 5.4. Figure 5.4 shows a computation over three processes. Clearly, $C(e_{11}) < C(e_{22})$ and $C(e_{11}) < C(e_{32})$. However, we can see from the figure that event e_{11} is causally related to event e_{22} but not to event e_{32}, since a path exists from e_{11} to e_{22} but not from e_{11} to e_{32}. Note that the initial clock values are assumed to be zero and d of equations 5.1 and 5.2 is assumed to equal 1. In other words, in Lamport's system of clocks, we can guarantee that if $C(a) < C(b)$ then $b \not\rightarrow a$ (i.e., the future cannot influence the past), however, we cannot say whether events a and b are causally related or not (i.e., whether there exists a path between a and b that moves only forward along the time axis in the space-time diagram) by just looking at the timestamps of the events.

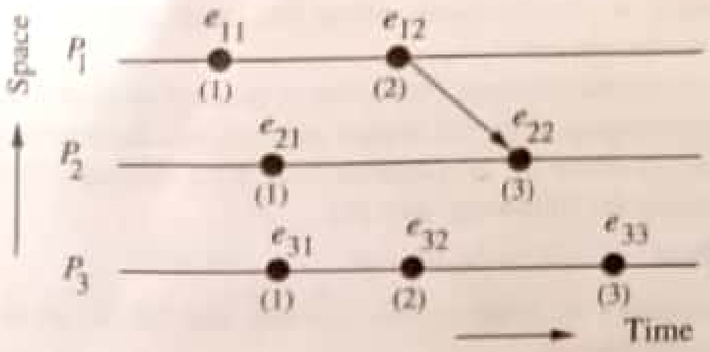

FIGURE 5.4
A space-time diagram.

In the system of vector clocks,

$$a \rightarrow b \quad \text{iff} \quad t^a < t^b \tag{5.5}$$

Thus, the system of vector clocks allows us to order events and decide whether two events are causally related or not by simply looking at the timestamps of the events. If we know the processes where the events occur, the above test can be further simplified (see Problem 5.1). Note that an event e can causally affect another event e' (events e_{12} and e_{22} in Fig. 5.5) if there exists a path that propagates the (local) time knowledge of event e to event e' [16].

In the next section, we present an application of vector clocks for the causal ordering of messages.

5.5 CAUSAL ORDERING OF MESSAGES

The causal ordering of messages was first proposed by Birman and Joseph [1] and was implemented in ISIS. The causal ordering of messages deals with the notion of maintaining the same causal relationship that holds among "message send" events with the corresponding "message receive" events. In other words, if $Send(M_1) \rightarrow Send(M_2)$ (where $Send(M)$ is the event sending message M), then every recipient of both messages M_1 and M_2 must receive M_1 before M_2. The causal ordering of messages should not be confused with the causal ordering of events, which deals with the notion of causal relationship among the events. In a distributed system, the causal ordering of messages is not automatically guaranteed. For example, Fig. 5.6 shows a violation of causal ordering of messages in a distributed system. In this example, $Send(M_1) \rightarrow Send(M_2)$. However, $M2$ is delivered before $M1$ to process P_3. (The numbers circled indicate the correct causal order to deliver messages.)

Techniques for the causal ordering of messages are useful in developing distributed algorithms and may simplify the algorithms themselves. For example, for applications such as replicated database systems, it is important that every process in charge of updating a replica receives the updates in the same order to maintain the consistency of the database [1]. In the absence of causal ordering of messages, each and every update must be checked to ensure that it does not violate the consistency constraints.

We next describe two protocols that make use of vector clocks for the causal ordering of messages in distributed systems. The first protocol is implemented in ISIS [2], wherein the processes are assumed to communicate using broadcast messages. The

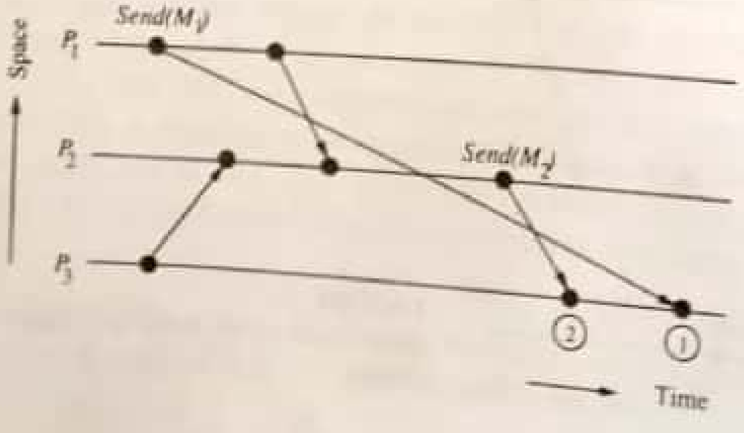

FIGURE 5.6
An example of the violation of causal ordering of messages.

second protocol does not require processes to communicate only through broadcast messages. Both protocols require that the messages be delivered reliably (lossless and uncorrupted).

BASIC IDEA. The basic idea of both the protocols is to deliver a message to a process only if the message immediately preceding it has been delivered to the process. Otherwise, the message is not delivered immediately but is buffered until the message immediately preceding it is delivered. A vector accompanying each message contains the necessary information for a process to decide whether there exists a message preceding it.

BIRMAN-SCHIPER-STEPHENSON PROTOCOL

1. Before broadcasting a message m, a process P_i increments the vector time $VT_{P_i}[i]$ and timestamps m. Note that $(VT_{P_i}[i] - 1)$ indicates how many messages from P_i precede m.
2. A process $P_j \neq P_i$, upon receiving message m timestamped VT_m from P_i, delays its delivery until both the following conditions are satisfied.
   a. $VT_{P_j}[i] = VT_m[i] - 1$
   b. $VT_{P_j}[k] \geq VT_m[k]\ \forall k \in \{1, 2, \ldots, n\} - \{i\}$
   where n is the total number of processes.
   Delayed messages are queued at each process in a queue that is sorted by vector time of the messages. Concurrent messages are ordered by the time of their receipt.
3. When a message is delivered at a process P_j, VT_{P_j} is updated according to the vector clocks rule IR2 (see Eq. 5.4).

Step 2 is the key to the protocol. Step 2(a) ensures that process P_j has received all the messages from P_i that preceed m. Step 2(b) ensures that P_j has received all those messages received by P_i before sending m. Since the event ordering relation "$\rightarrow$" imposed by vector clocks is acyclic, the protocol is deadlock free.

The Birman-Schiper-Stephenson causal ordering protocol requires that the processes communicate through broadcast messages. We next describe a protocol proposed by Schiper, Eggli, and Sandoz [20], which does not require processes to communicate only by broadcast messages.