CHAPTER 16

Distributed Operating Systems

Learning Objectives

After reading this chapter, you should be able to:

- Describe a distributed operating system and how it is different from network operation systems.
- Explain the difficulties in designing and developing distributed systems.
- Discuss the many useful distributed applications such as logical clock, leader election, and consensus.
- Explain fault-tolerant computing, and challenges in developing fault-tolerant systems.

16.1 Introduction

In many systems such as banking, telephony, airline reservation, flight control, industrial process control, etc., data and functions are spatially distributed. For example, a bank may have many computers installed at various branches that are geographically spread across distances. Some branches may have sophisticated resources that other branches lack. Another thing is that a user may like to withdraw money from any branch office even though she holds accounts at a particular branch. There is a great need to connect these computers for the purpose of resource sharing by all branches. Connecting together these computers enables users to access data stored at one branch from other branches, and thereby helps to eliminate or reduce redundancies of data and functions.

Distributed systems are a necessity in modern day life. We will become even more dependent on them in coming days. A distributed system provides an environment in which users can conveniently use resources residing anywhere in the system. In this chapter we will study some basic issues (such as interprocess communications, deadlocks, fault-tolerance, etc.) in designing

≫ A distributed system is a powerful paradigm envisioned in the history of computing to make several computers working together to solve user problems. It is essentially a system with multiple computers interconnected by computer networks that are intended to work cooperatively.

and developing distributed systems. We will also study some distributed computation problems and their solutions.

16.2 Distributed Systems

A *distributed system* is a well-knit collection of independent-, autonomous computers, connected together via a communications network. These computers do not share memory, I/O devices, or system clocks. A computer network, in a restricted way, is often referred to as a distributed system. A computer in the network is called variously *site*, *host*, *node*, etc. A site may be a uniprocessor or multi-processor computer, and is a full fledged computer system, that is typically managed by a local operating system.[1] The sites work concurrently, and communicate among themselves by exchanging messages over the network. Each site is fitted with network interface cards and supporting software for the purpose of communications with other sites on the network (see Section 10.6 on page 295). The messages are exchanged via communication lines. The message exchange is also asynchronous; that is, messages may be delivered after arbitrary delays.

Figure 16.1 presents a model of a typical computer network. There are four sites that are connected by five communication lines. All sites are directly connected to one another, except sites A and C. Message exchange between sites A and C has to be routed via sites B and/or D.

Distributed systems are more often referred to as *loosely coupled systems*, in contrast to tightly coupled multiprocessor systems. One of the key differences between loosely coupled and tightly coupled systems is that in the latter there is only one operating system shared by all the processors, but in a distributed system usually there is one operating system for each site and different sites can have different kinds of operating systems.

A distributed system promotes resource sharing, expedites user computations, and enhances system availability and reliability. It allows incremental system growth by adding or replacing an individual computer. Users at one site are able to access resources at other sites. They may be able to break up their computational tasks, and utilize multiple processors available at different sites to speed up computations. Availability of resources is another feature a distributed system can provide: If a few sites fail, the remaining sites can continue their operations and provide services to users.

There were many attempts to develop operating systems to facilitate distributed processing in distributed systems. The earlier operating systems developed for distributed systems were called network operating systems which later evolved to include various levels of transparencies (such as location transparency, access transparency, control transparency, data transparency, execution transparency, name transparency, migration transparency, network transparency, performance transparency, etc.) in the system. Such systems are referred to as distributed operating systems.

≫ In real networks, messages from one site to another may hop many intermediate sites before reaching their destinations.

≫ Traditionally, *distributed system* is a term used to refer a large array of computer systems, ranging from tightly coupled systems connected by switching networks to loosely coupled systems connected by computer networks such as local area network, wide area network, etc. In this book, distributed systems refer only to loosely coupled systems.

[1] With respect to a site, resources residing at that site are "local" to the site, and resources residing at other sites are "remote" to the site.

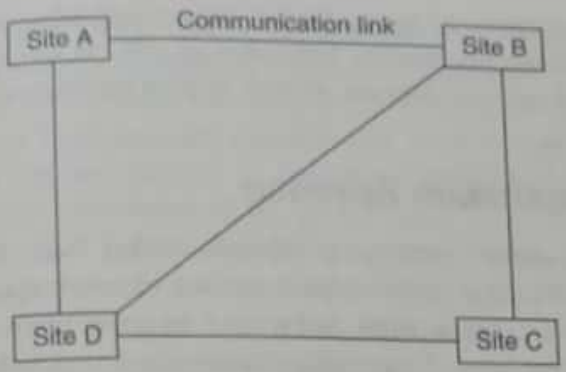

Figure 16.1: *A simple computer network.*

The concept of a distributed operating system was very ambitious in the beginning. For the users it must look and feel like an ordinary centralized operating system, but runs on several independent computers connected by networks. That is, the network, the computers, and their management must be completely transparent to the users. The (distributed) operating system must create the illusion that everything is done locally in one system. Although it was apparent that building such a truly distributed operating system is almost impossible, there were numerous attempts to build one. In this book, we look at some basic issues related to distributed systems and study solutions proposed in those and related efforts.

≫ The term transparency in the context of distributed computing means that the system should hide its "distributed nature" from its applications and users by creating the illusion of a normal centralized system. For example, the location transparency must assure that the users and applications should not have to be aware of the physical location of the resources.

16.3 Goals and Challenges

Distributed systems are a necessity in modern life. We will become even more dependent on them in coming days. They have been developed for many different reasons: resource sharing, load balancing, electronic communications, fault-tolerance, high availability, etc. Users should be able to develop and run distributed applications. The following subsections briefly discuss these topics.

16.3.1 Resource Sharing

Resource sharing is the prime goal in developing distributed systems. Users at one site can conveniently use resources available at other sites. For example, users at site S_1 can draw pictures on a graphics plotter available at site S_2. Users at site S_2 can make use of various storage devices available at site S_1. Many organizations such as banking systems have data distributed geographically. Such data are vital resources for the survival of these organizations. A site may need to refer to data stored at other sites to make various decisions. Distributed databases have become very common in recent years.

≫ Railways, airlines, hotel reservation systems are examples of distributed databases.

16.3.2 Load Balancing

For a given distributed system, workloads at different sites vary with time. If some sites are highly overloaded with work, parts of computations from these

≫ There is an analogy in our day-to-day life. When someone has too much work to do, she seeks out people to help her to finish the work.

≫ Reliability is also a major concern in non-distributed systems where devices holding information may be physically damaged.

sites may be transferred to lightly loaded sites to normalize workload among all sites. Load balancing enables busy sites to offload some work to lightly loaded sites, and thereby improve overall system performance.

16.3.3 Computation Speedup

Using several resources (such as processors, disks) from different sites in parallel can significantly improve performance of some applications. Users may partition their computation tasks, and execute these partitions concurrently at different sites. This enables them reduce response times for their overall computation. Note the users, and not the system, partition their tasks.

16.3.4 Electronic Communications

Users at various sites are able to interact in real-time. They can chat, browse web pages, exchange electronic mails, etc. The communications are done without physically exchange of hardware devices such as CDs, sending in parcels through postal mail.

16.3.5 Availability and Fault-tolerance

Two important goals of a distributed system are enhanced availability of information (data and functions) to users, with higher reliability. Availability ensures that information is accessible when it is needed; it is the percentage of time the system is up and accessible to users. Reliability ensures that the system does not corrupt or lose information. It is of utmost concern in a distributed system; but a highly reliable system that is poorly available is hardly of any use.

A distributed system may suffer from many kinds of failures (e.g., site or link failures, message loss or corruption) at unpredictable times. Development of distributed systems that are tolerant to certain kinds of failures is important. A *fault-tolerant system* is one that functions smoothly, with graceful degradation, when some failures occur in the system. Fault-tolerance is a system's ability to behave in a well-defined manner when faults do occur in the system. Such systems are said to be reliable and available. Note that reliability and correctness are two different concepts. A system is correct as long as it is free of faults and its internal data structures do not contain any error. A system is reliable if failures do not seriously impair its satisfactory operation. We quote from Peter Denning here: "Reliability means not freedom from faults and errors, but tolerance against them."

≫ An asynchronous exchange of messages means that messages may be delivered after arbitrary delays.

16.3.6 Design and Development Challenges

Computers in a distributed system do not share memory or have common I/O devices. They communicate among themselves only by exchanging messages. Computations as well as message communications are asynchronous.

Relative process execution speeds and message transmission delays are both unknown, and often, unbounded. Algorithms for asynchronous distributed systems should not rely on such bounds for their correctness. These pose a severe challenge in design and development of distributed systems. A fundamental problem encountered in distributed systems is that no computer may have perfect knowledge of the global state of the entire system. Such lack of "up-to-date" information by computers makes implementation of many features much harder or impossible. For example, distributed deadlock detection, load balancing, and resource management are difficult problems. Managing global resources without accurate state information is very difficult. For example, there are no online perfect load balancing algorithms. Practical systems employ some kind of heuristic algorithms to rationalize workload among computers. Fault-tolerance is another formidable challenge. Constructing fault-tolerant communication primitives to exchange messages among different sites is of utmost importance.

≫ "You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done." – Leslie Lamport.

16.4 Computer Networks

The backbone of a distributed system is the computer network. At the lower level of a distributed system is a collection of computers that are connected through a communications network. We can connect the computers in various configurations. This section discusses some well-known network configurations.

16.4.1 Hardware Requirements

To build a distributed system, we have to form a network of computers at the lower level. We need to connect computers to one another so that they can exchange information. Each computer is fitted with one or more network interface cards that are connected to those of other computers over Ethernet cables, telephone lines, fibre channels, wireless, and/or satellite links. Examples of communication devices are Ethernet card, blue tooth, router, etc. We discussed Ethernet cards in Section 10.6. We, in this book, do not discuss protocols of network communications, that is, how messages from one site are transferred to another.

16.4.2 Network Topology

Computers in a network are connected to one another through a communications network, and the network can be configured in many different ways. A network may be *fully connected*, where every site is directly connected to every other site in the network. Otherwise, a network is *partially connected*. Most practical networks are partially connected to save the cost of setting up fully connected networks. Figure 16.1 on page 439 is an example of a partially connected network. Other examples of partially connected networks having some regular topological structures are shown in Fig. 16.2. Network topologies are displayed as graphs, where boxes are sites, and edges are communication links. The edges are usually bidirectional, but they can also

≫ There are physical limitations in building connected large netwo

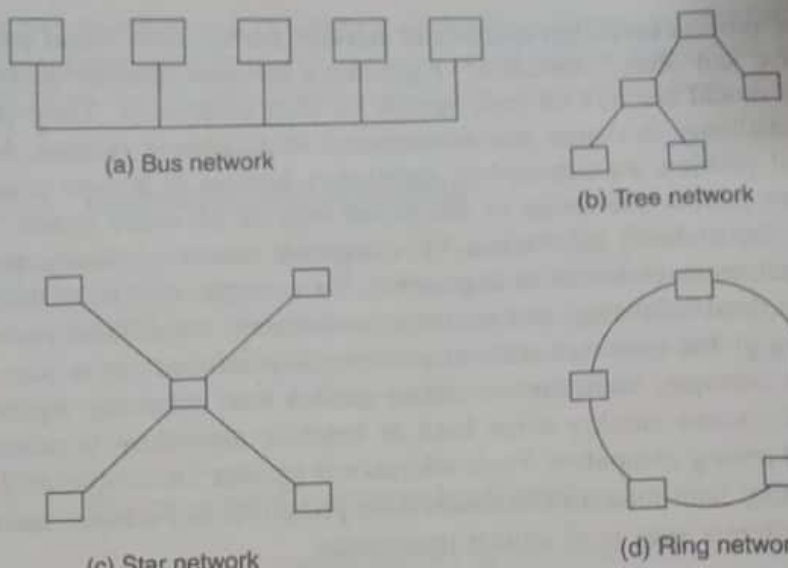

Figure 16.2: *Typical computer networks.*

be unidirectional. In a partially connected network, there may not be a direct communication link between two sites. Consequently, a message from one site to another may need to travel through intermediate sites. A partially connected network may not be as available as a fully connected network. Failure of a few sites or lines may partition the network into many disconnected subnetworks. In Fig. 16.2(c), the failure of the central site leaves the other four disconnected from one another.

Computer networks are classified into two broad categories: *local-area networks* (LANs) and *wide-area networks* (WANs).

Local-area Networks

A LAN consists of computers that are distributed over small geographical areas, such as a single building or a small campus within the radius of a few hundred meters. The entire network is normally under a single management. Communication links have higher speeds and lower error rates than of WANs. Commonly used links are twisted pairs and fibre optic cables. Typical Ethernet cables have speeds of 10 megabits per second to 10 gigabits per second. LANs can be connected to one another through gateways.

Wide-area Networks

A WAN consists of computers that are distributed over large geographical areas, such as a state or an entire country. Communication links have lower speeds and higher error rates of LANs. Telephone lines, microwave links, satellite channels are used as WAN links. These links are normally controlled by special purpose communication processors. A WAN provides communication infrastructure. Host computers reside on LANs that are connected to WANs gateways.

16.5 Interprocess Communication

In order to achieve some common tasks, processes need to communicate with one another. Shared memory-based communications are not possible between processes residing at different sites as sites do not share main memory. In such situations, processes can communicate only by exchanging messages. Thereby, good message communication primitives are vital in distributed systems. The underlying communications network transports messages from one site to another. (In this book we do not discuss how the network transports messages from one site to another.) The operating system collects messages from local processes and puts the messages into the communications network, and collects messages from the network and delivers them to local processes.

16.5.1 Communication Primitives

The message communication model that is widely used in distributed systems is the *client-server model*, in which a sender process (called client) that needs some service sends a message to another process (called server). Processes (clients or servers) do not need to know details of communications network to send and/or receive messages to and from one another. They only need to know the way to identify themselves to the system. In a single computer system, *pid*s (process identification numbers, see Section 4.7 on page 93) are used to identify processes. In addition, processes in a distributed system must be able to locate the hosts of one another. For interprocess communications, we must have a means to connect two remote processes. Broadly speaking, there are two basic ways to connect two processes: direct and indirect communications.

» In simple terms, clients ask questions and servers answer them.

1. Each process is identified by a pair ⟨*hname*, *pid*⟩. The pair is called a *process name*, where the *hname* is a unique name (or network number) of a host in the network, and the *pid* is the process's local process identification number within the host. The message communication is direct. A sender process knows the identity of the receiver to whom it is sending a message.
2. Alternatively, each host implements a set of communication points or ports. A communication point is represented by a pair ⟨*hname*, *id*⟩ where the *hname* is a unique name (or network number) of a host in the network, and the *id* is a unique port number within the host. The message communication is indirect. A sender process may not know the identity of the receiver process. The sender sends messages to a specific port, and any process that is attached to the port receives the messages.

In either scheme, we assume that given a process name or a port name, the communications network can locate the host where the process or the port resides. The operating system implements *send* and *receive* communication primitives. The *send* primitive specifies a destination (a process name or a communication port) and provides a message. The *receive* primitive tells from whom (a process or a port) it expects a message and provides a buffer where the incoming message is stored.

≫ Non-blocking primitives allow more concurrency, but programming with them becomes difficult. Irreproducible, timing dependent errors are often very difficult to diagnose and debug.

The send and receive primitives, though their specifications are very simple, raised a lot of controversies in the earlier phase of implementation—mostly, about what should be the acceptable semantics of these primitives. Two fundamental aspects that must be taken into account are (1) unreliable- versus reliable execution and (2) non-blocking- versus blocking execution.

An unreliable send operation puts a message in the network and returns to the caller. There is no guarantee that the message will be delivered to the destination; no automatic message retransmission is attempted if the original message is lost. A reliable send operation handles lost messages and retransmissions internally so that when the send invocation returns to the caller the message has been delivered to the destination. When a send operation is non-blocking, the send returns control to the sender as soon as the message data is queued for subsequent transmission. When the message is actually transmitted, the sender is interrupted to inform about the transmission. A blocking send does not return control to the sender until the message has been sent (for unreliable systems) or until the message has been delivered to the destination (for reliable systems).

In reliable receive, the receiver sends an acknowledgment to the sender. In unreliable receive, the receiver normally does not send an acknowledgment to the sender. A blocking receive does not return control to the receiver until a message has been copied into the receive buffer. A non-blocking receive returns immediately if there is no message for the receiver.

≫ In the Internet domain, for TCP based sockets, the communication is reliable; for UDP based sockets, the communication is unreliable. Sockets are used to implement a client-server architecture of interprocess communications. Servers listen to well-known ports to which clients connect.

16.5.2 Sockets

A socket is a mechanism of implementing both reliable- and unreliable interprocess communications in distributed systems. It is somewhat similar to the pipe interprocess communication scheme (see Section 6.4.4 on page 145). A pipe is used for unidirectional byte streams between two related processes in the same computer system, while a socket can be used for bidirectional communications between two unrelated processes that can reside in two different computer systems, see Fig. 16.3. A socket is part of a logical communication channel, and represents one end point of the channel. Thus, there is one socket at either end of the channel. It is an indirect communication scheme: any process that has access to the socket can exchange data over the channel. Each socket has a host address that is used to identify the host having the socket. The address format depends on the communication domain of the socket. In the Internet domain, a socket address is a pair consisting of a 32-bit host number and a 32-bit port number. Other communication domains have different socket structures.

≫ In UNIX, socket descriptors lie in the same name space as file descriptors are. Internally, a socket descriptor points to a socket object instead of a file object.

The operating system implements a set of system calls for sockets. Here we describe some system calls that are implemented in UNIX systems. The socket system call creates a new socket and returns a socket descriptor. The client executes the connect system call, through a socket descriptor, to initiate a connection with another (possibly, remote) socket. A remote process (called *server* or *daemon*) also creates a socket and binds the socket to a (well-known) local address. Then, the server executes the listen system call on the socket to

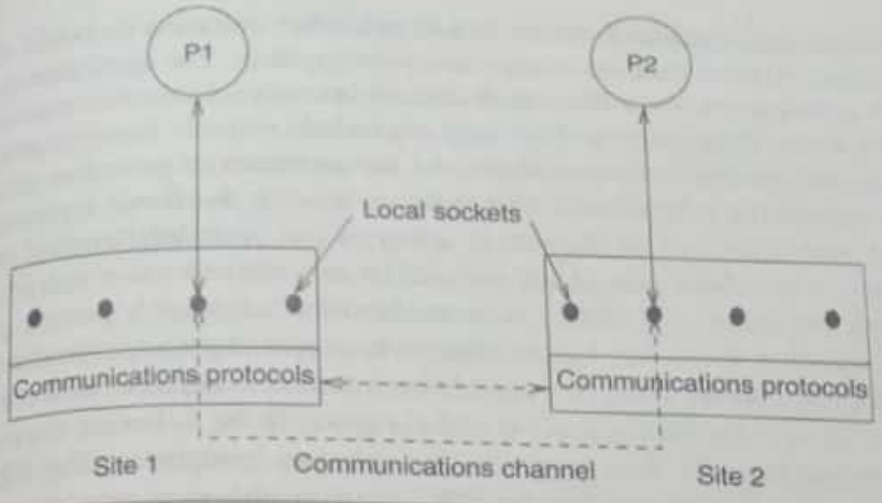

Figure 16.3: *Socket connection between two processes P1 and P2 in two sites.*

inform the operating system that it is ready to accept connections from other processes (*clients*). The server then executes the accept system call to accept individual connection requests from clients. The accept system call returns a new socket descriptor for a newly accepted connection. The server, at this point, usually creates a new process or thread, and goes back to the original socket descriptor to accept further new connections. The newly created process/thread serves the client whose connection request has been accepted by the server. An application process executes ordinary file-system-based read and write system calls on a socket descriptor to receive and send, respectively, messages over the socket connection. Finally, a close system call on a socket descriptor terminates the connection.

≫ In UNIX systems, for the Internet domain, port numbers less than 1024 are reserved for servers and are well-known. For example, HTTP servers listen to port 80, telnet servers to port 23. The /etc/services file contains well-known services and their ports.

16.5.3 Remote Procedure Call

It is convenient to have interprocess communication schemes such as socket primitives available to application programs. The socket primitives are used to set up communication channels between two processes, residing on the same computer or on two different computers. However, most application developers wish to execute functions/procedures that reside outside the process address space in another process also possibly residing at a remote site.

Remote procedure abstraction is useful for providing communications across a network. It helps application developers execute remote[2] functions as if the functions are in the same address space. The abstraction makes the semantics of interprocess communications as simple as possible to the local procedure/function calls, the way traditional function calls transfer control and data within a program running in a single address space. The application may not know the exact location of the (remote) function. The system takes care of all the

≫ Underneath the socket abstraction, the operating system implements a set of communication protocols that handle data transport requests through sockets. The protocols help processes to exchange data without concern about the underlying communications network. Ultimately, the operating system executes network interface driver routines to transmit- and receive data to and from remote computers (see Section 10.6).

[2]Here remote means outside the process address space, and the process need not be on a different computer system.

≫ PRC is a message-passing IPC scheme. But the messages are transparent to applications. RPC has been successfully used in many distributed applications.

communications required for the remote procedure execution on behalf of applications without their knowledge or involving them. The application developers do not worry about the complexities of interprocess communications, and, unlike socket programming, they need not include codes in the applications to handle interprocess communications. As the semantics of procedure calls are well understood, it becomes a little easier to develop distributed applications.

A procedure call mechanism is a way of one procedure/function to call another, where both procedures are part of one address space and process. Remote procedure call (RPC) is a mechanism that helps a process call a remote routine; it is a way for one process to execute a program (procedure or function) in another process (possibly on a remote computer) as if the program is local to the calling process address space. In the following discussion, we assume that the two processes are on two computers. The RPC is implemented through running an RPC daemon on every computer. Each daemon listens to a well-known local RPC port. A daemon contains the required routine that it executes and sends back the output of the execution to the requesting process. When an application process makes an RPC, it is blocked until it gets back a reply. The procedure parameters are passed across the network to the site where the procedure is executed. When the execution of the procedure is complete, the results are passed back to the calling process and the process resumes its local execution.

≫ Calling a function/procedure/method by passing parameters is a basic step in programming. Using such a widely known programming technique for remote communication alleviates the complexity of distributed communication. One of the main objectives of remote procedure calls is that the users can make the remote communications similar to local procedure calls and need not deal with details of message communication at the network level.

The RPC is normally implemented as follows: the calling program (called a client) makes a normal procedure call $p(x, y, z, \ldots)$ as if p is a procedure in the client address space. A dummy or stub procedure p' is included in the client address space or dynamically linked to it upon the call, see Fig. 16.4; the stub is generated by a compiler or pre-processor and not supplied by applications. The local stub collects the parameters of the procedure, forms them into one or more messages in a standard format, and sends the messages to the remote computer having the actual procedure. (The

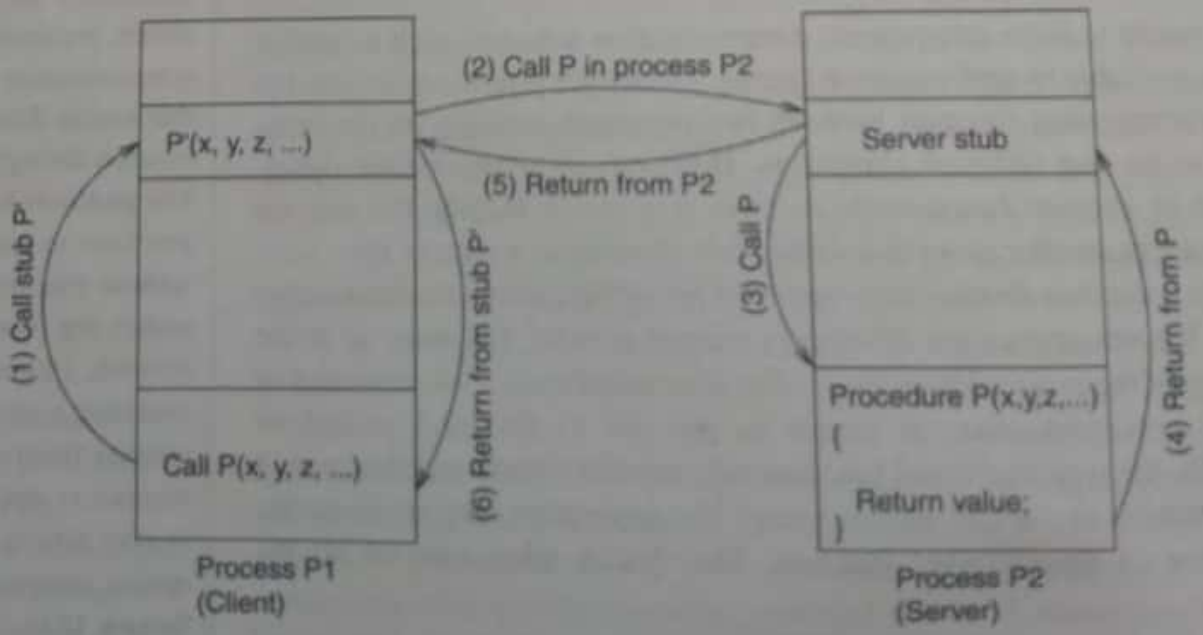

Figure 16.4: *Steps in a RPC call.*

stub finds out where the called function is located.) It then blocks the client, and waits for an answer from the remote computer. At the remote computer, there is another stub procedure in which the RPC server process waits for messages. Upon receiving messages, the receiver stub unpacks the parameters from the message, and makes a call to the local procedure p with those parameters. The result of the procedure invocation follows an analogous path in the reverse direction.

There is one restriction in RPCs, and that is in parameter passing. Most programming languages support parameter passing by value and by reference. Passing parameters by their values is easier; the stub copies the values in messages. Passing reference (pointer) parameters is not so easy. We need, for each object, a unique system-wide pointer so that the object can be remotely accessed. Such accesses add steep overhead. Value parameters also sometimes create problems in their representations across different types of processor architectures, for example little endian versus big endian integer values, different bit-sizes for the same data type, etc.

Developing a fault-tolerant RPC mechanism is a demanding task. Error handling in distributed systems is far more complex than in centralized systems. There are several things that could go wrong during an RPC call. If a client makes an RPC when the server is dead, the client may be left blocked forever unless a timeout scheme is built into the client- or local operating system. Timeouts introduce a new problem where the client timeouts too quickly assuming that the server is down, when actually the server is congested and responding to requests very slowly. If the network is congested or has disconnected the client from the server, the client cannot tell whether or not the server is down. Crashing clients may also cause troubles for the server if it expects input data from clients.

The crucial question is what should be the accepted semantics of RPC during failures. Ideally we would expect exactly one execution of the remote procedure, but it is probably impossible to achieve that. Some systems offer no guarantee at all (zero or more executions), some guarantee at the most one execution (zero or one), and some guarantee at least one execution (one or more). We will not discuss these issues in this book.

16.6 System Environment

A single computer system creates an environment (i.e., software platform) in which users do their work with relative ease. Creating such an environment is one of the major goals of operating systems. Likewise, a distributed system creates an environment in which users can use with relative ease all the resources (local or remote) available throughout the system. Users may or may not be aware of the existence of multiple computers in the system, and thereby the system can be a network operating system or a true distributed operating system, accordingly. We discuss these two types of systems in the next two subsections. The key issue in these two kinds of systems is how aware users are of the multiplicity of computers. This visibility occurs in three important spheres, namely program execution, file system, and protection.

16.6.2 Distributed Operating Systems

A *distributed operating system* is one that looks and feels to its users like an ordinary (single computer) operating system but runs on multiple-, independent computers. It appears to users to be a single virtual uniprocessor computer system. The computers each run a part of the global, system-wide operating system. The multiplicity of computers is hidden, that is, they are invisible to end users. In this environment, users access remote resources in the same way as if the resources are local. Data- or computation migration from one computer to another is transparently done under the control of operating systems, and not explicitly done by users themselves.

It is the operating system that determines whether a network of computers is a true distributed system. Users should not know on which computers their programs run or where their files (programs and data) reside. The

≫ When a user wants to run an application, she manually chooses the computer for the run in case of a network operating system. In a true distributed operating system, the system automatically chooses the computer.

[1]Newcastle Connection is a software subsystem that can be added to each of a set of physically interconnected UNIX or UNIX look-alike systems so as to construct a distributed system which is functionally indistinguishable at both the user- and the program level from a conventional single processor UNIX system.

has indeed

16.7.7 Distributed File Systems

A distributed file system is a collection of independent (local) file systems, where an individual file system resides in a single computer. A distributed file system should appear to be a single centralized file system to applications. Multiplicity of file storage devices and remote file systems must be transparent to applications. Applications should not distinguish between local- and remote files. They access files by location-independent filenames; in fact, they may not know the exact locations of files. They use files by making system calls to the respective local computers. The file management system does the mapping between filenames to computers where the files reside. To enhance availability and performance, the management system may even transparently replicate files. File replicas reside on failure-independent computers. Failure of one replica should not cause failures of other replicas.

≫ The only non-transparent action in NFS is that the mount operation takes the address of the remote computer as a parameter.

Network file system (NFS) from Sun Microsystems is an example of location-transparent distributed file system. NFS permits mounting of remote directories on local file tree. Any remote directory can be mounted on any local directory. Once mounted, the remote directory appears as a sub-tree of the local file tree, as usual hiding what is under the original local mount

directory. Users on the local computer can access any file on the mounted sub-tree in a location-transparent manner.

16.7.8 Deadlock Detection

We have discussed deadlock issues in Section 7.6 on page 186. Each site can periodically check whether there are local deadlocks, and take actions accordingly. The major problem is detecting deadlocks that involve processes from multiple sites. Consider a simple example, as depicted in Fig. 16.9. We have two resources r_1 and r_2 at two different sites s_1 and s_2, respectively. A process P_1 running at site s_1 holds r_1, and another process P_2 at s_2 holds r_2. Then P_1 tries to acquire r_2 through a proxy process P_1' at s_2, and P_2 tries to acquire r_1 through a proxy P_2' at s_1. Site s_1 (respectively, s_2) does not know what process P_1 (respectively, P_2) is doing at the other site. There is no local deadlock at either site, but there is a global deadlock that goes like this: P_2' is blocked for r_1 held by P_1 that is blocked on P_1' that is blocked for r_2 held by P_2 that is blocked on P_2'. Normally each site maintains a wait-for graph for local processes and those remote (proxy) processes that access local resources. An individual site does not have complete information about remote processes. Absence of deadlocks at individual sites does not preclude absence of deadlocks in the entire system, as depicted in Fig. 16.9. To determine a global deadlock, we need to merge all local wait-for graphs to form a single wait-for graph.

» There are, however, some difficulties in using location-independent file systems. Each computer system usually has special files; for example, device files are in /dev, binary executable in /bin. How do we unify all device files under the single /dev directory? It is not easy to answer this question.

Collecting coherent wait-for graphs from different sites is not easy. Communication delays make it difficult to get an accurate view of the (global) state of the entire system. Consequently, a newly formed deadlock may not be detected immediately or a deadlock indicated may no longer exist. The latter is called *phantom deadlock*. A deadlock is a stable property: once formed, it remains there until it is broken; it will be eventually detected and broken. But, phantom deadlocks are of some concern because we may needlessly abort- and rollback some processes.

Like in many distributed applications, there are two ways to solve the deadlock detection problem. In a centralized deadlock detection algorithm, there is a deadlock coordinator. The coordinator communicates with all sites to construct a single (global) wait-for graph involving all the processes and resources in

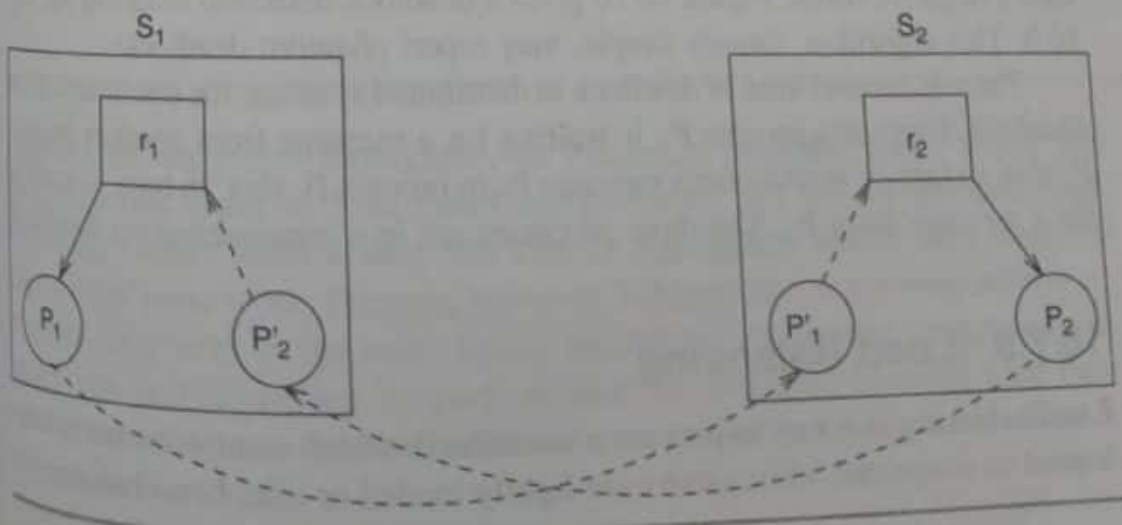

Figure 16.9: *A typical tributed deadlock.*

16.8 Fault Tolerance

Constructing a distributed system in theory is always easier than constructing it in practice. We envisage many challenges in constructing practical distributed systems. The real difficulty comes from component failures. Distributed systems are subject to many kinds of failures (such as site failure, link failure, message corruption, message loss) at unpredictable times. A *failure* is an event at which a system violates its specification; that is, observed behaviour deviates from specified behaviour. A failure is caused by an error in the system. Errors do not always produce failures, because some errors are expunged by error recovery algorithms. Errors are produced by mechanical- or algorithmic defects called *faults*. That is, a fault may cause an error that, in turn, may lead to a failure. We need to make distributed systems as fault-tolerant as possible so that simple faults do not lead to system-wide failures.

Sites may exhibit two types of failures: stopping failure and Byzantine failure. For stopping failure, a site can fail by stopping any where in the middle of its execution with or without warning. For Byzantine failure, a site can behave arbitrarily as if it is a rogue site. A link can fail by losing messages intermittently or by stopping.

It may not be possible to tolerate all kinds of faults. A distributed system must be tolerant of some kinds of faults. Informally, fault-tolerance of a system is the ability of the system to behave in a well-defined manner once faults occur. That is, even though a few faults may lead to component failures, the failures must not impair the entire system. For example, if one

site fails, the remaining sites should be able to continue their operations without collapsing the entire system. If some sites are down and copies of the failed resources are available at other sites, then the system as a whole should still be available for users to continue their work in gracefully degraded manner. Otherwise, the degradation is in both performance and function availability. The goal is to have a system that works fast in the normal case, and works gracefully in case of failures.

16.8.1 Redundancy

No matter how well a distributed system is designed, there is always a possibility of failure in the system if faults are too severe. To tolerate faults, one must employ some form of redundancy in the system. Redundancy means that something is present that is not needed during normal operations, but the redundant parts acquire great value if faults occur in the system. The redundancy appears in the form of both space and time. Space redundancy means that there are additional hardware and software that are used when faults occur. Time redundancy means that, when faults occur, the system carries out additional actions to circumvent the faults.

≫ If there is only one instance of some resource and the resource fails, it cannot be available to provide service. We need multiple copies of resources to achieve fault-tolerance. Redundancy is a natural requirement toward building fault-tolerant systems.

16.8.2 Detection and Recovery

If one wants to continue with normal operations in the presence of faults, we do require a distributed system to have mechanisms to detect, diagnose, and correct failures without involving users; and if correction is not possible, to reconfigure itself in the event of failures. Detection refers to discovering that a fault exists. It is a part of system redundancy. Diagnostics refer to identifying the location of faults. Repair or reconfiguration refers, respectively, to either fixing the fault or returning the system into a consistent state by isolating faulty units. Note that the detection- and repair mechanisms must themselves be fault-tolerant. Because of the lack of global knowledge in a distributed system, it is sometimes difficult to determine the kind of failures. For example, most failures manifest as message loss at non-faulty sites. Upon detecting a failure, the system needs to take some corrective actions to sustain the functioning of the remaining active components.

Normally, neighbouring sites are in constant touch with one another. They exchange (*I-am-alive* kind of) messages at regular intervals. If a site does not receive such messages from a neighbour, then some failure may have occurred in the system, and the site informs other sites on the network that a direct link between it and the so-and-so neighbour has broken, and the system starts reconfiguring itself. This is achieved by isolating the failed components from the system until they are repaired. Likewise when a broken link is repaired or a failed site recovers, the neighbouring site(s) broadcast the information to the others on the network. Error detection and correction codes are used for message transmission, and timeout events for lost messages.