CHAPTER 1

Fundamentals

1.1 WHAT IS A DISTRIBUTED COMPUTING SYSTEM?

Over the past two decades, advancements in microelectronic technology have resulted in the availability of fast, inexpensive processors, and advancements in communication technology have resulted in the availability of cost-effective and highly efficient computer networks. The net result of the advancements in these two technologies is that the price-performance ratio has now changed to favor the use of interconnected, multiple processors in place of a single, high-speed processor.

Computer architectures consisting of interconnected, multiple processors are basically of two types:

1. *Tightly coupled systems.* In these systems, there is a single systemwide primary memory (address space) that is shared by all the processors [Fig. 1.1(a)]. If any processor writes, for example, the value 100 to the memory location x, any other processor subsequently reading from location x will get the value 100. Therefore, in these systems, any communication between the processors usually takes place through the shared memory.

2. *Loosely coupled systems.* In these systems, the processors do not share memory, and each processor has its own local memory [Fig. 1.1(b)]. If a processor writes the value

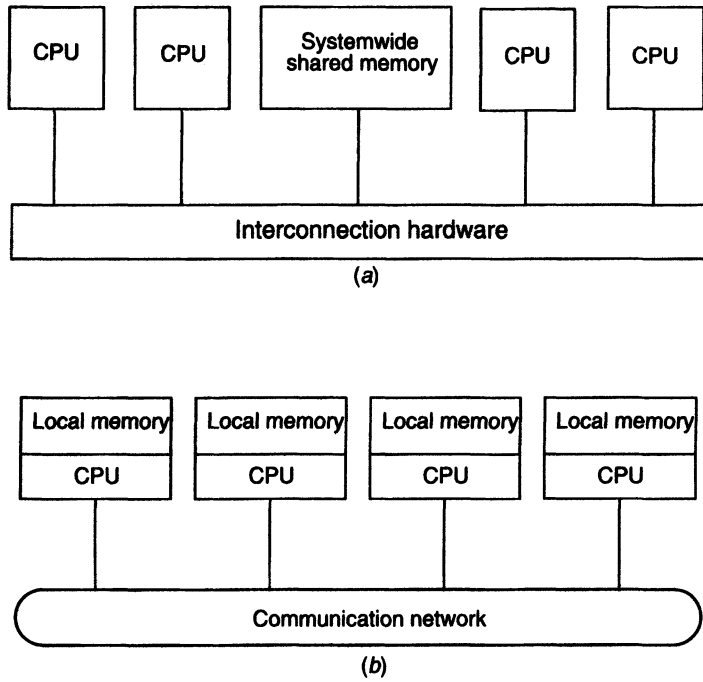


Fig. 1.1 Difference between tightly coupled and loosely coupled multiprocessor systems: (*a*) a tightly coupled multiprocessor system; (*b*) a loosely coupled multiprocessor system.

100 to the memory location x, this write operation will only change the contents of its local memory and will not affect the contents of the memory of any other processor. Hence, if another processor reads the memory location *x*, it will get whatever value was there before in that location of its own local memory. In these systems, all physical communication between the processors is done by passing messages across the network that interconnects the processors.

Usually, tightly coupled systems are referred to as *parallel processing systems*, and loosely coupled systems are referred to as *distributed computing systems*, or simply distributed systems. In this book, however, the term "distributed system" will be used only for true distributed systems—distributed computing systems that use distributed operating systems (see Section 1.5). Therefore, before the term "true distributed system" is defined in Section 1.5, the term "distributed computing system" will be used to refer to loosely coupled systems. In contrast to the tightly coupled systems, the processors of distributed computing systems can be located far from each other to cover a wider geographical area. Furthermore, in tightly coupled systems, the number of processors that can be usefully deployed is usually small and limited by the bandwidth of the shared memory. This is not the case with distributed computing systems that are more freely expandable and can have an almost unlimited number of processors.

In short, a distributed computing system is basically a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals, and the communication between any two processors of the system takes place by message passing over the communication network. For a particular processor, its own resources are *local*, whereas the other processors and their resources are *remote*. Together, a processor and its resources are usually referred to as a *node* or *site* or *machine* of the distributed computing system.

1.2 EVOLUTION OF DISTRIBUTED COMPUTING SYSTEMS

Early computers were very expensive (they cost millions of dollars) and very large in size (they occupied a big room). There were very few computers and were available only in research laboratories of universities and industries. These computers were run from a console by an operator and were not accessible to ordinary users. The programmers would write their programs and submit them to the computer center on some media, such as punched cards, for processing. Before processing a job, the operator would set up the necessary environment (mounting tapes, loading punched cards in a card reader, etc.) for processing the job. The job was then executed and the result, in the form of printed output, was later returned to the programmer.

The job setup time was a real problem in early computers and wasted most of the valuable central processing unit (CPU) time. Several new concepts were introduced in the 1950s and 1960s to increase CPU utilization of these computers. Notable among these are batching together of jobs with similar needs before processing them, automatic sequencing of jobs, off-line processing by using the concepts of buffering and spooling, and multiprogramming. *Batching* similar jobs improved CPU utilization quite a bit because now the operator had to change the execution environment only when a new batch of jobs had to be executed and not before starting the execution of every job. *Automatic job sequencing* with the use of control cards to define the beginning and end of a job improved CPU utilization by eliminating the need for human job sequencing. *Off-line processing* improved CPU utilization by allowing overlap of CPU and input/output (I/O) operations by executing those two actions on two independent machines (I/O devices are normally several orders of magnitude slower than the CPU). Finally, *multiprogramming* improved CPU utilization by organizing jobs so that the CPU always had something to execute.

However, none of these ideas allowed multiple users to directly interact with a computer system and to share its resources simultaneously. Therefore, execution of interactive jobs that are composed of many short actions in which the next action depends on the result of a previous action was a tedious and time-consuming activity. Development and debugging of programs are examples of interactive jobs. It was not until the early 1970s that computers started to use the concept of *time sharing* to overcome this hurdle. Early time-sharing systems had several dumb terminals attached to the main computer. These terminals were placed in a room different from the main computer room. Using these terminals, multiple users could now simultaneously execute interactive jobs and share the resources of the computer system. In a time-sharing system, each user is given

mathematical computations. In a centralized system, the users have to perform all types of computations on the only available computer. However, a distributed computing system may have a pool of different types of computers, in which case the most appropriate one can be selected for processing a user's job depending on the nature of the job. For instance, we saw that in a distributed computing system that is based on the hybrid model, interactive jobs can be processed at a user's own workstation and the processors in the pool may be used to process noninteractive, computation-intensive jobs.

Note that the advantages of distributed computing systems mentioned above are not achieved automatically but depend on the careful design of a distributed computing system. This book deals with the various design methodologies that may be used to achieve these advantages.

1.5 WHAT IS A DISTRIBUTED OPERATING SYSTEM?

Tanenbaum and Van Renesse [1985] define an *operating system* as a program that controls the resources of a computer system and provides its users with an interface or virtual machine that is more convenient to use than the bare machine. According to this definition, the two primary tasks of an operating system are as follows:

1. To present users with a virtual machine that is easier to program than the underlying hardware.
2. To manage the various resources of the system. This involves performing such tasks as keeping track of who is using which resource, granting resource requests, accounting for resource usage, and mediating conflicting requests from different programs and users.

Therefore, the users' view of a computer system, the manner in which the users access the various resources of the computer system, and the ways in which the resource requests are granted depend to a great extent on the operating system of the computer system. The operating systems commonly used for distributed computing systems can be broadly classified into two types—*network operating systems* and *distributed operating systems*. The three most important features commonly used to differentiate between these two types of operating systems are system image, autonomy, and fault tolerance capability. These features are explained below.

1. *System image.* The most important feature used to differentiate between the two types of operating systems is the image of the distributed computing system from the point of view of its users. In case of a network operating system, the users view the distributed computing system as a collection of distinct machines connected by a communication subsystem. That is, the users are aware of the fact that multiple computers are being used. On the other hand, a distributed operating system hides the existence of multiple computers and provides a single-system image to its users. That is, it makes a collection

of networked machines act as a *virtual uniprocessor*. The difference between the two types of operating systems based on this feature can be best illustrated with the help of examples. Two such examples are presented below.

In the case of a network operating system, although a user can run a job on any machine of the distributed computing system, he or she is fully aware of the machine on which his or her job is executed. This is because, by default, a user's job is executed on the machine on which the user is currently logged. If the user wants to execute a job on a different machine, he or she should either log on to that machine by using some kind of "remote login" command or use a special command for remote execution to specify the machine on which the job is to be executed. In either case, the user knows the machine on which the job is executed. On the other hand, a distributed operating system dynamically and automatically allocates jobs to the various machines of the system for processing. Therefore, a user of a distributed operating system generally has no knowledge of the machine on which a job is executed. That is, the selection of a machine for executing a job is entirely manual in the case of network operating systems but is automatic in the case of distributed operating systems.

With a network operating system, a user is generally required to know the location of a resource to access it, and different sets of system calls have to be used for accessing local and remote resources. On the other hand, users of a distributed operating system need not keep track of the locations of various resources for accessing them, and the same set of system calls is used for accessing both local and remote resources. For instance, users of a network operating system are usually aware of where each of their files is stored and must use explicit *file transfer* commands for moving a file from one machine to another, but the users of a distributed operating system have no knowledge of the location of their files within the system and use the same command to access a file irrespective of whether it is on the local machine or on a remote machine. That is, control over file placement is done manually by the users in a network operating system but automatically by the system in a distributed operating system.

Notice that the key concept behind this feature is "transparency." We will see later in this chapter that a distributed operating system has to support several forms of transparency to achieve the goal of providing a single-system image to its users. Moreover, it is important to note here that with the current state of the art in distributed operating systems, this goal is not fully achievable. Researchers are still working hard to achieve this goal.

2. *Autonomy.* A network operating system is built on a set of existing centralized operating systems and handles the interfacing and coordination of remote operations and communications between these operating systems. That is, in the case of a network operating system, each computer of the distributed computing system has its own local operating system (the operating systems of different computers may be the same or different), and there is essentially no coordination at all among the computers except for the rule that when two processes of different computers communicate with each other, they must use a mutually agreed on communication protocol. Each computer functions independently of other computers in the sense that each one makes independent decisions about the creation and termination of their own processes and management of local

resources. Notice that due to the possibility of difference in local operating systems, the system calls for different computers of the same distributed computing system may be different in this case.

On the other hand, with a distributed operating system, there is a single systemwide operating system and each computer of the distributed computing system runs a part of this global operating system. The distributed operating system tightly interweaves all the computers of the distributed computing system in the sense that they work in close cooperation with each other for the efficient and effective utilization of the various resources of the system. That is, processes and several resources are managed globally (some resources are managed locally). Moreover, there is a single set of globally valid system calls available on all computers of the distributed computing system.

The set of system calls that an operating system supports are implemented by a set of programs called the *kernel* of the operating system. The kernel manages and controls the hardware of the computer system to provide the facilities and resources that are accessed by other programs through system calls. To make the same set of system calls globally valid, with a distributed operating system identical kernels are run on all the computers of a distributed computing system. The kernels of different computers often cooperate with each other in making global decisions, such as finding the most suitable machine for executing a newly created process in the system.

In short, it can be said that the degree of autonomy of each machine of a distributed computing system that uses a network operating system is considerably high as compared to that of machines of a distributed computing system that uses a distributed operating system.

3. *Fault tolerance capability.* A network operating system provides little or no fault tolerance capability in the sense that if 10% of the machines of the entire distributed computing system are down at any moment, at least 10% of the users are unable to continue with their work. On the other hand, with a distributed operating system, most of the users are normally unaffected by the failed machines and can continue to perform their work normally, with only a 10% loss in performance of the entire distributed computing system. Therefore, the fault tolerance capability of a distributed operating system is usually very high as compared to that of a network operating system.

The following definition of a distributed operating system given by Tanenbaum and Van Renesse [1985] covers most of its features mentioned above:

> A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs). The key concept here is transparency. In other words, the use of multiple processors should be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a "virtual uniprocessor," not as a collection of distinct machines. [P. 419].

A distributed computing system that uses a network operating system is usually referred to as a *network system*, whereas one that uses a distributed operating system is

usually referred to as a *true distributed system* (or simply a distributed system). In this book, the term *distributed system* will be used to mean a true distributed system.

Note that with the current state of the art in distributed operating systems, it is not possible to design a completely true distributed system. Completely true distributed systems are the ultimate goal of researchers working in the area of distributed operating systems.

1.6 ISSUES IN DESIGNING A DISTRIBUTED OPERATING SYSTEM

In general, designing a distributed operating system is more difficult than designing a centralized operating system for several reasons. In the design of a centralized operating system, it is assumed that the operating system has access to complete and accurate information about the environment in which it is functioning. For example, a centralized operating system can request status information, being assured that the interrogated component will not change state while awaiting a decision based on that status information, since only the single operating system asking the question may give commands. However, a distributed operating system must be designed with the assumption that complete information about the system environment will never be available. In a distributed system, the resources are physically separated, there is no common clock among the multiple processors, delivery of messages is delayed, and messages could even be lost. Due to all these reasons, a distributed operating system does not have up-to-date, consistent knowledge about the state of the various components of the underlying distributed system. Obviously, lack of up-to-date and consistent information makes many things (such as management of resources and synchronization of cooperating activities) much harder in the design of a distributed operating system. For example, it is hard to schedule the processors optimally if the operating system is not sure how many of them are up at the moment.

Despite these complexities and difficulties, a distributed operating system must be designed to provide all the advantages of a distributed system to its users. That is, the users should be able to view a distributed system as a virtual centralized system that is flexible, efficient, reliable, secure, and easy to use. To meet this challenge, the designers of a distributed operating system must deal with several design issues. Some of the key design issues are described below. The rest of the chapters of this book basically contain detailed descriptions of these design issues and the commonly used techniques to deal with them.

1.6.1 Transparency

We saw that one of the main goals of a distributed operating system is to make the existence of multiple computers invisible (transparent) and provide a single system image to its users. That is, a distributed operating system must be designed in such a way that a collection of distinct machines connected by a communication subsystem

appears to its users as a virtual uniprocessor. Achieving complete transparency is a difficult task and requires that several different aspects of transparency be supported by the distributed operating system. The eight forms of transparency identified by the International Standards Organization's Reference Model for Open Distributed Processing [ISO 1992] are access transparency, location transparency, replication transparency, failure transparency, migration transparency, concurrency transparency, performance transparency, and scaling transparency. These transparency aspects are described below.

Access Transparency

Access transparency means that users should not need or be able to recognize whether a resource (hardware or software) is remote or local. This implies that the distributed operating system should allow users to access remote resources in the same way as local resources. That is, the user interface, which takes the form of a set of system calls, should not distinguish between local and remote resources, and it should be the responsibility of the distributed operating system to locate the resources and to arrange for servicing user requests in a user-transparent manner.

This requirement calls for a well-designed set of system calls that are meaningful in both centralized and distributed environments and a global resource naming facility. We will see in Chapters 3 and 4 that due to the need to handle communication failures in distributed systems, it is not possible to design system calls that provide complete access transparency. However, the area of designing a global resource naming facility has been well researched with considerable success. Chapter 10 deals with the concepts and design of a global resource naming facility. The distributed shared memory mechanism described in Chapter 5 is also meant to provide a uniform set of system calls for accessing both local and remote memory objects. Although this mechanism is quite useful in providing access transparency, it is suitable only for limited types of distributed applications due to its performance limitation.

Location Transparency

The two main aspects of location transparency are as follows:

1. *Name transparency.* This refers to the fact that the name of a resource (hardware or software) should not reveal any hint as to the physical location of the resource. That is, the name of a resource should be independent of the physical connectivity or topology of the system or the current location of the resource. Furthermore, such resources, which are capable of being moved from one node to another in a distributed system (such as a file), must be allowed to move without having their names changed. Therefore, resource names must be unique systemwide.

2. *User mobility.* This refers to the fact that no matter which machine a user is logged onto, he or she should be able to access a resource with the same name. That is, the user should not be required to use different names to access the same resource from two

different nodes of the system. In a distributed system that supports user mobility, users can freely log on to any machine in the system and access any resource without making any extra effort.

Both name transparency and user mobility requirements call for a systemwide, global resource naming facility.

Replication Transparency

For better performance and reliability, almost all distributed operating systems have the provision to create replicas (additional copies) of files and other resources on different nodes of the distributed system. In these systems, both the existence of multiple copies of a replicated resource and the replication activity should be transparent to the users. That is, two important issues related to replication transparency are naming of replicas and replication control. It is the responsibility of the system to name the various copies of a resource and to map a user-supplied name of the resource to an appropriate replica of the resource. Furthermore, replication control decisions such as how many copies of the resource should be created, where should each copy be placed, and when should a copy be created/deleted should be made entirely automatically by the system in a user-transparent manner. Replica management issues are described in Chapter 9.

Failure Transparency

Failure transparency deals with masking from the users' partial failures in the system, such as a communication link failure, a machine failure, or a storage device crash. A distributed operating system having failure transparency property will continue to function, perhaps in a degraded form, in the face of partial failures. For example, suppose the file service of a distributed operating system is to be made failure transparent. This can be done by implementing it as a group of file servers that closely cooperate with each other to manage the files of the system and that function in such a manner that the users can utilize the file service even if only one of the file servers is up and working. In this case, the users cannot notice the failure of one or more file servers, except for slower performance of file access operations. Any type of service can be implemented in this way for failure transparency. However, in this type of design, care should be taken to ensure that the cooperation among multiple servers does not add too much overhead to the system.

Complete failure transparency is not achievable with the current state of the art in distributed operating systems because all types of failures cannot be handled in a user-transparent manner. For example, failure of the communication network of a distributed system normally disrupts the work of its users and is noticeable by the users. Moreover, an attempt to design a completely failure-transparent distributed system will result in a very slow and highly expensive system due to the large amount of redundancy required for tolerating all types of failures. The design of such a distributed system, although theoretically possible, is not practically justified.

Migration Transparency

For better performance, reliability, and security reasons, an object that is capable of being moved (such as a process or a file) is often migrated from one node to another in a distributed system. The aim of migration transparency is to ensure that the movement of the object is handled automatically by the system in a user-transparent manner. Three important issues in achieving this goal are as follows:

1. Migration decisions such as which object is to be moved from where to where should be made automatically by the system.
2. Migration of an object from one node to another should not require any change in its name.
3. When the migrating object is a process, the interprocess communication mechanism should ensure that a message sent to the migrating process reaches it without the need for the sender process to resend it if the receiver process moves to another node before the message is received.

Chapter 7 deals with the first issue. The second issue calls for a global resource naming facility, which is described in Chapter 10. Ways to handle the third issue are described in Chapter 8.

Concurrency Transparency

In a distributed system, multiple users who are spatially separated use the system concurrently. In such a situation, it is economical to share the system resources (hardware or software) among the concurrently executing user processes. However, since the number of available resources in a computing system is restricted, one user process must necessarily influence the action of other concurrently executing user processes, as it competes for resources. For example, concurrent update to the same file by two different processes should be prevented. Concurrency transparency means that each user has a feeling that he or she is the sole user of the system and other users do not exist in the system. For providing concurrency transparency, the resource sharing mechanisms of the distributed operating system must have the following four properties:

1. An event-ordering property ensures that all access requests to various system resources are properly ordered to provide a consistent view to all users of the system.

2. A mutual-exclusion property ensures that at any time at most one process accesses a shared resource, which must not be used simultaneously by multiple processes if program operation is to be correct.

3. A no-starvation property ensures that if every process that is granted a resource, which must not be used simultaneously by multiple processes, eventually releases it, every request for that resource is eventually granted.

4. A no-deadlock property ensures that a situation will never occur in which competing processes prevent their mutual progress even though no single one requests more resources than available in the system.

Chapter 6 deals with the above-mentioned issues of concurrency transparency.

Performance Transparency

The aim of performance transparency is to allow the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system. As far as practicable, a situation in which one processor of the system is overloaded with jobs while another processor is idle should not be allowed to occur. That is, the processing capability of the system should be uniformly distributed among the currently available jobs in the system.

This requirement calls for the support of intelligent resource allocation and process migration facilities in distributed operating systems. Chapters 7 and 8 deal with these two issues.

Scaling Transparency

The aim of scaling transparency is to allow the system to expand in scale without disrupting the activities of the users. This requirement calls for open-system architecture and the use of scalable algorithms for designing the distributed operating system components. Section 1.6.3 of this chapter and Section 2.6 of Chapter 2 focus on the issues of designing an open distributed system. On the other hand, since every component of a distributed operating system must use scalable algorithms, this issue has been dealt with in almost all chapters of the book.

1.6.2 Reliability

In general, distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources. However, the existence of multiple instances of the resources alone cannot increase the system's reliability. Rather, the distributed operating system, which manages these resources, must be designed properly to increase the system's reliability by taking full advantage of this characteristic feature of a distributed system.

A *fault* is a mechanical or algorithmic defect that may generate an error. A fault in a system causes system failure. Depending on the manner in which a failed system behaves, system failures are of two types—fail-stop [Schlichting and Schneider 1983] and Byzantine [Lamport et al. 1982]. In the case of *fail-stop failure*, the system stops functioning after changing to a state in which its failure can be detected. On the other hand, in the case of *Byzantine failure*, the system continues to function but produces wrong results. Undetected software bugs often cause Byzantine failure of a system. Obviously, Byzantine failures are much more difficult to deal with than fail-stop failures.

For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and

This functionality is sometimes hidden in language-level constructs. For example, RPC provides automatic message generation and reception according to a procedural specification. However, the basic communication paradigm remains the same because the communicating processes directly interact with each other for exchanging the shared data.

In contrast to the message-passing paradigm, the shared-memory paradigm provides to processes in a system with a shared address space. Processes use this address space in the same way they use normal local memory. That is, processes access data in the shared address space through the following two basic primitives, of course, with some variations in the syntax and semantics in different implementations:

data = **Read** (*address*)
Write (*address*, *data*)

Read returns the *data* item referenced by *address*, and *write* sets the contents referenced by *address* to the value of *data*.

We saw in Chapter 1 that the two major kinds of multiple-instruction, multiple-data-stream (MIMD) multiprocessors that have become popular and gained commercial acceptance are tightly coupled shared-memory multiprocessors and loosely coupled distributed-memory multiprocessors. The use of a shared-memory paradigm for interprocess communication is natural for distributed processes running on tightly coupled shared-memory multiprocessors. However, for loosely coupled distributed-memory systems, no physically shared memory is available to support the shared-memory paradigm for interprocess communication. Therefore, until recently, the interprocess communication mechanism in loosely coupled distributed-memory multiprocessors was limited only to the message-passing paradigm. But some recent loosely coupled distributed-memory systems have implemented a software layer on top of the message-passing communication system to provide a shared-memory abstraction to the programmers. The shared-memory abstraction gives these systems the illusion of physically shared memory and allows programmers to use the shared-memory paradigm. The software layer, which is used for providing the shared-memory abstraction, can be implemented either in an operating system kernel or in runtime library routines with proper system kernel support. The term Distributed Shared Memory (DSM) refers to the shared-memory paradigm applied to loosely coupled distributed-memory systems [Stumm and Zhou 1990].

As shown in Figure 5.1, DSM provides a virtual address space shared among processes on loosely coupled processors. That is, DSM is basically an abstraction that integrates the local memory of different machines in a network environment into a single logical entity shared by cooperating processes executing on multiple sites. The shared memory itself exists only virtually. Application programs can use it in the same way as a traditional virtual memory, except, of course, that processes using it can run on different machines in parallel. Due to the virtual existence of the shared memory, DSM is sometimes also referred to as *Distributed Shared Virtual Memory* (*DSVM*).

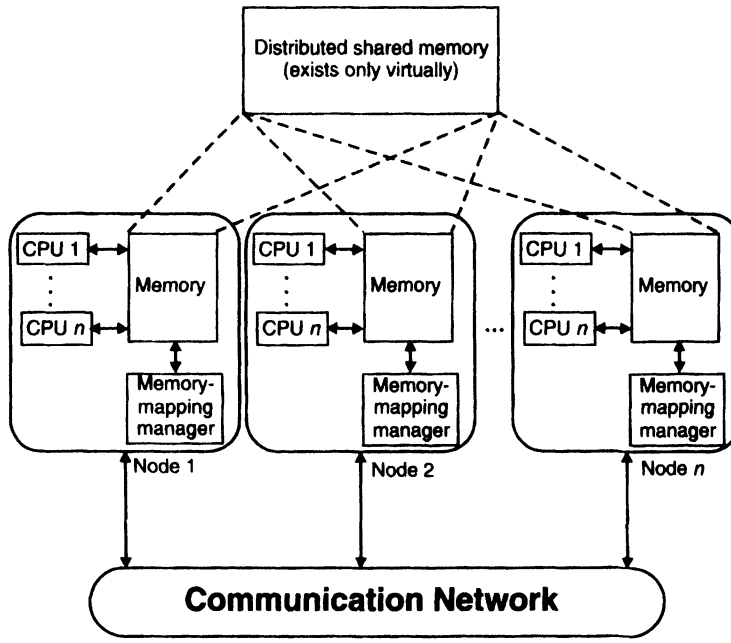


Fig. 5.1 Distributed shared memory (DSM).

5.2 GENERAL ARCHITECTURE OF DSM SYSTEMS

The DSM systems normally have an architecture of the form shown in Figure 5.1. Each node of the system consists of one or more CPUs and a memory unit. The nodes are connected by a high-speed communication network. A simple message-passing system allows processes on different nodes to exchange messages with each other.

The DSM abstraction presents a large shared-memory space to the processors of all nodes. In contrast to the shared physical memory in tightly coupled parallel architectures, the shared memory of DSM exists only virtually. A software memory-mapping manager routine in each node maps the local memory onto the shared virtual memory. To facilitate the mapping operation, the shared-memory space is partitioned into *blocks*.

Data caching is a well-known solution to address memory access latency. The idea of data caching is used in DSM systems to reduce network latency. That is, the main memory of individual nodes is used to cache pieces of the shared-memory space. The memory-mapping manager of each node views its local memory as a big cache of the shared-memory space for its associated processors. The basic unit of caching is a memory block.

When a process on a node accesses some data from a memory block of the shared-memory space, the local memory-mapping manager takes charge of its request. If the memory block containing the accessed data is resident in the local memory, the request is satisfied by supplying the accessed data from the local memory. Otherwise, a network block fault is generated and the control is passed to the operating system. The operating

CHAPTER 3

Message Passing

3.1 INTRODUCTION

A *process* is a program in execution. When we say that two computers of a distributed system are communicating with each other, we mean that two processes, one running on each computer, are in communication with each other. In a distributed system, processes executing on different computers often need to communicate with each other to achieve some common goal. For example, each computer of a distributed system may have a *resource manager* process to monitor the current status of usage of its local resources, and the resource managers of all the computers might communicate with each other from time to time to dynamically balance the system load among all the computers. Therefore, a distributed operating system needs to provide interprocess communication (IPC) mechanisms to facilitate such communication activities.

Interprocess communication basically requires information sharing among two or more processes. The two basic methods for information sharing are as follows:

1. Original sharing, or shared-data approach
2. Copy sharing, or message-passing approach

In the shared-data approach, the information to be shared is placed in a common memory area that is accessible to all the processes involved in an IPC. The shared-data

paradigm gives the conceptual communication pattern illustrated in Figure 3.1(*a*). On the other hand, in the message-passing approach, the information to be shared is physically copied from the sender process's address space to the address spaces of all the receiver processes, and this is done by transmitting the data to be copied in the form of messages (a *message* is a block of information). The message-passing paradigm gives the conceptual communication pattern illustrated in Figure 3.1(*b*). That is, the communicating processes interact directly with each other.

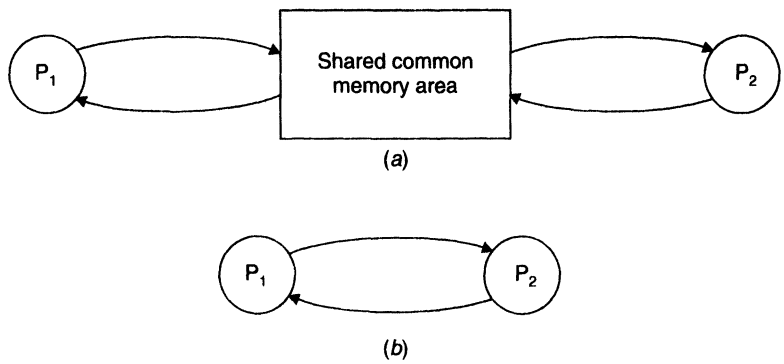


Fig. 3.1 The two basic interprocess communication paradigms: (*a*) The shared-data approach. (*b*) The message-passing approach.

Since computers in a network do not share memory, processes in a distributed system normally communicate by exchanging messages rather than through shared data. Therefore, message passing is the basic IPC mechanism in distributed systems.

A *message-passing system* is a subsystem of a distributed operating system that provides a set of message-based IPC protocols and does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers. It enables processes to communicate by exchanging messages and allows programs to be written by using simple communication primitives, such as *send* and *receive*. It serves as a suitable infrastructure for building other higher level IPC systems, such as remote procedure call (RPC; see Chapter 4) and distributed shared memory (DSM; see Chapter 5).

3.2 DESIRABLE FEATURES OF A GOOD MESSAGE-PASSING SYSTEM

3.2.1 Simplicity

A message-passing system should be simple and easy to use. It must be straightforward to construct new applications and to communicate with existing ones by using the primitives provided by the message-passing system. It should also be possible for a

programmer to designate the different modules of a distributed application and to send and receive messages between them in a way as simple as possible without the need to worry about the system and/or network aspects that are not relevant for the application level. Clean and simple semantics of the IPC protocols of a message-passing system make it easier to build distributed applications and to get them right.

3.2.2 Uniform Semantics

In a distributed system, a message-passing system may be used for the following two types of interprocess communication:

1. *Local communication*, in which the communicating processes are on the same node
2. *Remote communication*, in which the communicating processes are on different nodes

An important issue in the design of a message-passing system is that the semantics of remote communications should be as close as possible to those of local communications. This is an important requirement for ensuring that the message-passing system is easy to use.

3.2.3 Efficiency

Efficiency is normally a critical issue for a message-passing system to be acceptable by the users. If the message-passing system is not efficient, interprocess communication may become so expensive that application designers will strenuously try to avoid its use in their applications. As a result, the developed application programs would be distorted. An IPC protocol of a message-passing system can be made efficient by reducing the number of message exchanges, as far as practicable, during the communication process. Some optimizations normally adopted for efficiency include the following:

- Avoiding the costs of establishing and terminating connections between the same pair of processes for each and every message exchange between them
- Minimizing the costs of maintaining the connections
- Piggybacking of acknowledgment of previous messages with the next message during a connection between a sender and a receiver that involves several message exchanges

3.2.4 Reliability

Distributed systems are prone to different catastrophic events such as node crashes or communication link failures. Such events may interrupt a communication that was in progress between two processes, resulting in the loss of a message. A reliable IPC protocol can cope with failure problems and guarantees the delivery of a message. Handling of lost

messages usually involves acknowledgments and retransmissions on the basis of timeouts.

Another issue related to reliability is that of duplicate messages. Duplicate messages may be sent in the event of failures or because of timeouts. A reliable IPC protocol is also capable of detecting and handling duplicates. Duplicate handling usually involves generating and assigning appropriate sequence numbers to messages.

A good message-passing system must have IPC protocols to support these reliability features.

3.2.5 Correctness

A message-passing system often has IPC protocols for group communication that allow a sender to send a message to a group of receivers and a receiver to receive messages from several senders. Correctness is a feature related to IPC protocols for group communication. Although not always required, correctness may be useful for some applications. Issues related to correctness are as follows [Navratnam et al. 1988]:

- Atomicity
- Ordered delivery
- Survivability

Atomicity ensures that every message sent to a group of receivers will be delivered to either all of them or none of them. Ordered delivery ensures that messages arrive at all receivers in an order acceptable to the application. Survivability guarantees that messages will be delivered correctly despite partial failures of processes, machines, or communication links. Survivability is a difficult property to achieve.

3.2.6 Flexibility

Not all applications require the same degree of reliability and correctness of the IPC protocols. For example, in adaptive routing, it may be necessary to distribute the information regarding queuing delays in different parts of the network. A broadcast protocol could be used for this purpose. However, if a broadcast message is late in coming, due to communication failures, it might just as well not arrive at all as it will soon be outdated by a more recent one anyway. Similarly, many applications do not require atomicity or ordered delivery of messages. For example, a client may multicast a request message to a group of servers and offer the job to the first server that replies. Obviously, atomicity of message delivery is not required in this case. Thus the IPC protocols of a message-passing system must be flexible enough to cater to the various needs of different applications. That is, the IPC primitives should be such that the users have the flexibility to choose and specify the types and levels of reliability and correctness requirements of their applications. Moreover, IPC primitives must also have the flexibility to permit any kind of control flow between the cooperating processes, including synchronous and asynchronous *send/receive*.

3.2.7 Security

A good message-passing system must also be capable of providing a secure end-to-end communication. That is, a message in transit on the network should not be accessible to any user other than those to whom it is addressed and the sender. Steps necessary for secure communication include the following:

- Authentication of the receiver(s) of a message by the sender
- Authentication of the sender of a message by its receiver(s)
- Encryption of a message before sending it over the network

These issues will be described in detail in Chapter 11.

3.2.8 Portability

There are two different aspects of portability in a message-passing system:

1. The message-passing system should itself be portable. That is, it should be possible to easily construct a new IPC facility on another system by reusing the basic design of the existing message-passing system.

2. The applications written by using the primitives of the IPC protocols of the message-passing system should be portable. This requires that heterogeneity must be considered while designing a message-passing system. This may require the use of an external data representation format for the communications taking place between two or more processes running on computers of different architectures. The design of high-level primitives for the IPC protocols of a message-passing system should be done so as to hide the heterogeneous nature of the network.

3.3 ISSUES IN IPC BY MESSAGE PASSING

A message is a block of information formatted by a sending process in such a manner that it is meaningful to the receiving process. It consists of a fixed-length header and a variable-size collection of typed data objects. As shown in Figure 3.2, the header usually consists of the following elements:

- *Address.* It contains characters that uniquely identify the sending and receiving processes in the network. Thus, this element has two parts—one part is the sending process address and the other part is the receiving process address.
- *Sequence number.* This is the message identifier (ID), which is very useful for identifying lost messages and duplicate messages in case of system failures.

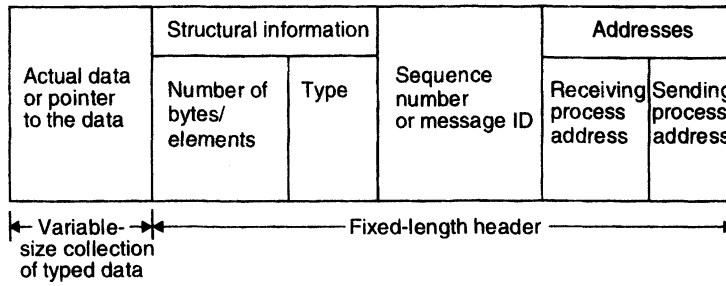


Fig. 3.2 A typical message structure.

- *Structural information.* This element also has two parts. The *type* part specifies whether the data to be passed on to the receiver is included within the message or the message only contains a pointer to the data, which is stored somewhere outside the contiguous portion of the message. The second part of this element specifies the length of the variable-size message data.

In a message-oriented IPC protocol, the sending process determines the actual contents of a message and the receiving process is aware of how to interpret the contents. Special primitives are explicitly used for sending and receiving the messages. Therefore, in this method, the users are fully aware of the message formats used in the communication process and the mechanisms used to send and receive messages.

In the design of an IPC protocol for a message-passing system, the following important issues need to be considered:

- Who is the sender?
- Who is the receiver?
- Is there one receiver or many receivers?
- Is the message guaranteed to have been accepted by its receiver(s)?
- Does the sender need to wait for a reply?
- What should be done if a catastrophic event such as a node crash or a communication link failure occurs during the course of communication?
- What should be done if the receiver is not ready to accept the message: Will the message be discarded or stored in a buffer? In the case of buffering, what should be done if the buffer is full?
- If there are several outstanding messages for a receiver, can it choose the order in which to service the outstanding messages?

These issues are addressed by the semantics of the set of communication primitives provided by the IPC protocol. A general description of the various ways in which these issues are addressed by message-oriented IPC protocols is presented below.

decimal form) from a user to identify the remote host with which the user wants to interact. However, as compared to numeric identifiers, symbolic names are easier for human beings to remember and use. Therefore, the Internet supports a scheme for the allocation and use of symbolic names for hosts and networks, such as *asuvax.eas.asu.edu* or *eas.asu.edu*. The named entities are called *domains* and the symbolic names are called *domain names*. The domain name space has a hierarchical structure that is entirely independent of the physical structure of the networks that constitute the Internet. A hierarchical naming scheme provides greater flexibility of name space management (described in detail in Chapter 10).

When domain names are accepted as parameters by application protocols such as FTP, TFTP, TELNET, SMTP, and so on, they must be translated to Internet addresses before making communication operation requests to lower level protocols. This job of mapping domain names to Internet addresses is performed by DNS. Further details of DNS are given in Chapter 10.

2.5.2 Protocols for Distributed Systems

Although the protocols mentioned previously provide adequate support for traditional network applications such as file transfer and remote login, they are not suitable for distributed systems and applications. This is mainly because of the following special requirements of distributed systems as compared to network systems [Kaashoek et al. 1993]:

- *Transparency.* Communication protocols for network systems use location-dependent process identifiers (such as port addresses that are unique only within a node). However, for efficient utilization of resources, distributed systems normally support process migration facility. With communication protocols for network systems, supporting process migration facility is difficult because when a process migrates, its identifier changes. Therefore, communication protocols for distributed systems must use location-independent process identifiers that do not change even when a process migrates from one node to another.
- *Client-server-based communication.* The communication protocols for network systems treat communication as an input/output device that is used to transport data between two nodes of a network. However, most communications in distributed systems are based on the client-server model in which a client requests a server to perform some work for it by sending the server a message and then waiting until the server sends back a reply. Therefore, communication protocols for distributed systems must have a simple, connectionless protocol having features to support request/response behavior.
- *Group communication.* Several distributed applications benefit from group communication facility that allows a message to be sent reliably from one sender to n receivers. Although many network systems provide mechanisms to do broadcast or multicast at the data-link layer, their communication protocols often hide these useful capabilities from the applications. Furthermore, although broadcast can be done by sending n point-to-point messages and waiting for n acknowledgments, this algorithm is inefficient and wastes bandwidth. Therefore,

communication protocols for distributed systems must support more flexible and efficient group communication facility in which a group address can be mapped on one or more data-link addresses and the routing protocol can use a data-link multicast address to send a message to all the receivers belonging to the group defined by the multicast address.

- *Security.* Security is a critical issue in networks, and encryption is the commonly used method to ensure security of message data transmitted across a network. However, encryption is expensive to use, and all nodes and all communication channels of a network are not untrustworthy for a particular user. Therefore, encryption should be used only when there is a possibility of a critical message to travel via an untrustworthy node/channel from its source node to the destination node. Hence a communication protocol is needed that can support a flexible and efficient encryption mechanism in which a message is encrypted only if the path it takes across the network cannot be trusted. Existing communication protocols for network systems do not provide such flexibility.
- *Network management.* Network management activities, such as adding/removing a node from a network, usually require manual intervention by a system administrator to update the configuration files that reflect the current configuration of the network. For example, the allocation of new addresses is often done manually. Ideally, network protocols should automatically handle network management activities to reflect dynamic changes in network configuration.
- *Scalability.* A communication protocol for distributed systems must scale well and allow efficient communication to take place in both LAN and WAN environments. A single communication protocol must be usable on both types of networks.

Two communication protocols that have been designed to achieve higher throughput and/or fast response in distributed systems and to address one or more of the issues stated above are *VMTP* (*Versatile Message Transport Protocol*) and *FLIP* (*Fast Local Internet Protocol*). VMPT provides group communication facility and implements a secure and efficient client-server-based communication protocol [Cheriton and Williamson 1989]. On the other hand, FLIP is designed to support transparency, efficient client-server-based communication, group communication, secure communication, and easy network management [Kaashoek et al. 1993]. These protocols are briefly described next.

The Versatile Message Transport Protocol

This is a transport protocol that has been especially designed for distributed operating systems and has been used in the V-System [Cheriton 1988]. It is a connectionless protocol that has special features to support request/response behavior between a client and one or more server processes. It is based on the concept of a message transaction that consists of a request message sent by a client to one or more servers followed by zero or more response messages sent back to the client by the servers, at most one per server. Most message transactions involve a single request message and a single response message.

Nonreplicated, Migrating Blocks

In this strategy each block of the shared memory has a single copy in the entire system. However, each access to a block causes the block to migrate from its current node to the node from where it is accessed. Therefore, unlike the previous strategy in which the owner node of a block always remains fixed, in this strategy the owner node of a block changes as soon as the block is migrated to a new node (Fig. 5.4). When a block is migrated away, it is removed from any local address space it has been mapped into. Notice that in this strategy only the processes executing on one node can read or write a given data item at any one time. Therefore the method ensures sequential consistency.

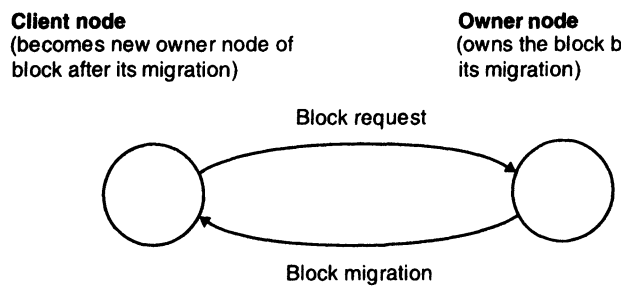


Fig. 5.4 Nonreplicated, migrating blocks (NRMB) strategy.

The method has the following advantages [Stumm and Zhou 1990]:

1. No communication costs are incurred when a process accesses data currently held locally.
2. It allows the applications to take advantage of data access locality. If an application exhibits high locality of reference, the cost of data migration is amortized over multiple accesses.

However, the method suffers from the following drawbacks:

1. It is prone to thrashing problem. That is, a block may keep migrating frequently from one node to another, resulting in few memory accesses between migrations and thereby poor performance.
2. The advantage of parallelism cannot be availed in this method also.

Data Locating in the NRMB Strategy. In the NRMB strategy, although there is a single copy of each block, the location of a block keeps changing dynamically. Therefore, one of the following methods may be used in this strategy to locate a block:

1. *Broadcasting.* In this method, each node maintains an *owned blocks table* that contains an entry for each block for which the node is the current owner (Fig. 5.5). When

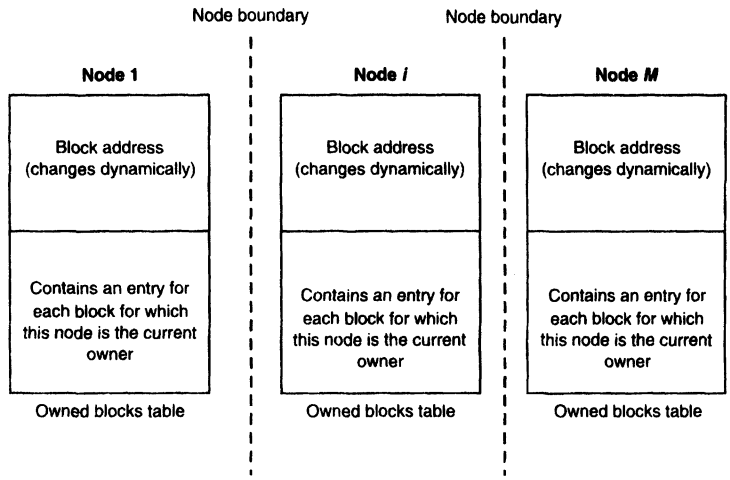


Fig. 5.5 Structure and locations of owned blocks table in the broadcasting data-locating mechanism for NRMB strategy.

a fault occurs, the fault handler of the faulting node broadcasts a read/write request on the network. The node currently having the requested block then responds to the broadcast request by sending the block to the requesting node.

A major disadvantage of the broadcasting algorithm is that it does not scale well. When a request is broadcast, not just the node that has the requested block but all nodes must process the broadcast request. This makes the communication subsystem a potential bottleneck. The network latency of a broadcast may also require accesses to take a long time to complete.

2. *Centralized-server algorithm.* In this method, a centralized server maintains a block table that contains the location information for all blocks in the shared-memory space (Fig. 5.6). The location and identity of the centralized server is well known to all nodes.

When a fault occurs, the fault handler of the faulting node (N) sends a request for the accessed block to the centralized server. The centralized server extracts the location information of the requested block from the block table, forwards the request to that node, and changes the location information in the corresponding entry of the block table to node N. On receiving the request, the current owner transfers the block to node N, which becomes the new owner of the block.

The centralized-server method suffers from two drawbacks: (a) the centralized server serializes location queries, reducing parallelism, and (b) the failure of the centralized server will cause the DSM system to stop functioning.

3. *Fixed distributed-server algorithm.* The fixed distributed-server scheme is a direct extension of the centralized-server scheme. It overcomes the problems of the centralized-server scheme by distributing the role of the centralized server. Therefore, in this scheme, there is a block manager on several nodes, and each block manager is given a

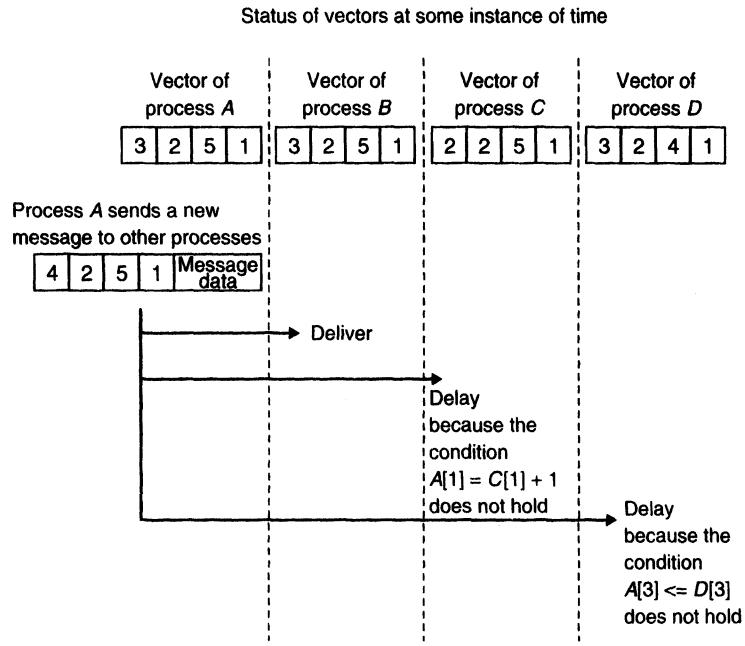


Fig. 3.18 An example to illustrate the CBCAST protocol for implementing causal ordering semantics.

3.11 CASE STUDY: 4.3BSD UNIX IPC MECHANISM

The socket-based IPC of the 4.3BSD UNIX system illustrates how a message-passing system can be designed using the concepts and mechanisms presented in this chapter. The system was produced by the Computer Systems Research Group (CSRG) of the University of California at Berkeley and is the most widely used and well documented message-passing system.

3.11.1 Basic Concepts and Main Features

The IPC mechanism of the 4.3BSD UNIX provides a general interface for constructing network-based applications. Its basic concepts and main features are as follows:

1. It is network independent in the sense that it can support communication networks that use different sets of protocols, different naming conventions, different hardware, and so on. For this, it uses the notion of *communication domain*, which refers to a standard set of communication properties. In this chapter we have seen that there are different methods of naming a communication endpoint. We also have seen that there are different semantics of communication related to synchronization, reliability, ordering, and so on. Different networks often use different naming conventions for naming communication endpoints

and possess different semantics of communication. These properties of a network are known as its *communication properties*. Networks with the same communication properties belong to a common communication domain (or protocol family). By providing the flexibility to specify a communication domain as a parameter of the communication primitive used, the IPC mechanism of the 4.3BSD UNIX allows the users to select a domain appropriate to their applications.

2. It uses a unified abstraction, called *socket*, for an endpoint of communication. That is, a socket is an abstract object from which messages are sent and received. The IPC operations are based on socket pairs, one belonging to each of a pair of communicating processes that may be on the same or different computers. A pair of sockets may be used for unidirectional or bidirectional communication between two processes. A message sent by a sending process is queued in its socket until it has been transmitted across the network by the networking protocol and an acknowledgment has been received (only if the protocol requires one). On the receiver side, the message is queued in the receiving process's socket until the receiving process makes an appropriate system call to receive it.

Any process can create a socket for use in communication with another process. Sockets are created within a communication domain. A created socket exists until it is explicitly closed or until every process having a reference to it exits.

3. For location transparency, it uses a two-level naming scheme for naming communication endpoints. That is, a socket can be assigned a high-level name that is a human-readable string. The low-level name of a socket is communication-domain dependent. For example, it may consist of a local port number and an Internet address. For translation of high-level socket names to their low-level names, 4.3BSD provides functions for application programs rather than placing the translation functions in the kernel. Note that a socket's high-level name is meaningful only within the context of the communication domain in which the socket is created.

4. It is highly flexible in the sense that it uses a typing mechanism for sockets to provide the semantic aspects of communication to applications in a controlled and uniform manner. That is, all sockets are typed according to their communication semantics, such as ordered delivery, unduplicated delivery, reliable delivery, connectionless communication, connection-oriented communication, and so on. The system defines some standard socket types and provides the flexibility to the users to define and use their own socket types when needed. For example, a socket of type *datagram* models potentially unreliable, connectionless packet communication, and a socket of type *stream* models a reliable connection-based byte stream.

5. Messages can be broadcast if the underlying network provides broadcast facility.

3.11.2 The IPC Primitives

The primitives of the 4.3BSD UNIX IPC mechanism are provided as system calls implemented as a layer on top of network communication protocols such as TCP, UDP, and so on. Layering the IPC mechanism directly on top of network communication

protocols helps in making it efficient. The most important available IPC primitives are briefly described below.

s = socket (domain, type, protocol)

When a process wants to communicate with another process, it must first create a socket by using the *socket* system call. The first parameter of this call specifies the communication domain. The most commonly used domain is the Internet communication domain because a large number of hosts in the world support the Internet communication protocols. The second parameter specifies the socket type that is selected according to the communication semantics requirements of the application. The third parameter specifies the communication protocol (e.g., TCP/IP or UDP/IP) to be used for the socket's operation. If the value of this parameter is specified as zero, the system chooses an appropriate protocol. The *socket* call returns a descriptor by which the socket may be referenced in subsequent system calls. A created socket is discarded with the normal *close* system call.

bind (s, addr, addrlen)

After creating a socket, the receiver must bind it to a socket address. Note that if two-way communication is desired between two processes, both processes have to receive messages, and hence both must separately bind their sockets to a socket address. The *bind* system call is used for this purpose. The three parameters of this call are the descriptor of the created socket, a reference to a structure containing the socket address to which the socket is to be bound, and the number of bytes in the socket address. Once a socket has been bound, its address cannot be changed.

It might seem more reasonable to combine the system calls for socket creation and binding a socket to a socket address (name) in a single system call. There are two main reasons for separating these two operations in different system calls. First, with this approach a socket can be useful without names. Forcing users to name every socket that is created causes extra burden on users and may lead to the assignment of meaningless names. Second, some communication domains might require additional, nonstandard information (such as type of service) for binding of a name to a socket. The need to supply this information at socket creation time will further complicate the interface.

connect (s, server_addr, server_addrlen)

The two most commonly used communication types in the 4.3BSD UNIX IPC mechanism are connection-based (stream) communication and connectionless (datagram) communication. In connection-based communication, two processes first establish a connection between their pairs of sockets. The connection establishment process is asymmetric because one of the processes keeps waiting for a request for a connection and the other makes a request for a connection. Once connection has been established, data can be transmitted between the two processes in either direction. This type of communication is useful for implementing client-server applications. A server creates a socket, binds a name

to it, and makes the name publicly known. It then waits for a connection request from client processes. Clients send connection requests to the server. Once the connection is established, they can exchange request and reply messages. Connection-based communication supports reliable exchange of messages.

In connectionless communication, a socket pair is identified each time a communication is made. For this, the sending process specifies its local socket descriptor and the socket address of the receiving process's socket each time it sends a message. Connectionless communication is potentially unreliable.

The *connect* system call is used in connection-based communication by a client process to request a connection establishment between its own socket and the socket of the server process with which it wants to communicate. The three parameters of this call are the descriptor of the client's socket, a reference to a structure containing the socket address of the server's socket, and the number of bytes in the socket address. The *connect* call automatically binds a socket address (name) to the client's socket. Hence prior binding is not needed.

listen (s, backlog)

The *listen* system call is used in case of connection-based communication by a server process to listen on its socket for client requests for connections. The two parameters of this call are the descriptor of the server's socket and the maximum number of pending connections that should be queued for acceptance.

snew = accept (s, client_addr, client_addrlen)

The *accept* system call is used in a connection-based communication by a server process to accept a request for a connection establishment made by a client and to obtain a new socket for communication with that client. The three parameters of this call are the descriptor of the server's socket, a reference to a structure containing the socket address of the client's socket, and the number of bytes in the socket address. Note that the call returns a descriptor (*snew*) that is the descriptor of a new socket that is automatically created upon execution of the *accept* call. This new socket is paired with the client's socket so that the server can continue to use the original socket with descriptor *s* for accepting further connection requests from other clients.

Primitives for Sending and Receiving Data

A variety of system calls are available for sending and receiving data. The four most commonly used are:

nbytes = read (snew, buffer, amount)
write (s, "message," msg_length)
amount = recvfrom (s, buffer, sender_address)
sendto (s, "message," receiver_address)

The *read* and *write* system calls are most suitable for use in connection-based communication. The *write* operation is used by a client to send a message to a server. The socket to be used for sending the message, the message, and the length of the message are specified as parameters to the call. The *read* operation is used by the server process to receive the message sent by the client. The socket of the server to which the client's socket is connected and the buffer for storing the received message are specified as parameters to the call. The call returns the actual number of characters received. The socket connection establishment between the client and the server behaves like a channel of stream data that does not contain any message boundary indications. That is, the sender pumps data into the channel and the receiver reads them in the same sequence as written by the corresponding write operations. The channel size is limited by a bounded queue at the receiving socket. The sender blocks if the queue is full and the receiver blocks if the queue is empty.

On the other hand, the *recvfrom* and *sendto* system calls are most suitable for use in case of connectionless communication. The *sendto* operation is used by a sender to send a message to a particular receiver. The socket through which the message is to be sent, the message, and a reference to a structure containing the socket address of the receiver to which the message is to be sent are specified as parameters to this call. The *recvfrom* operation is used by a receiver to receive a message from a particular sender. The socket through which the message is to be received, the buffer where the message is to be stored, and a reference to a structure containing the socket address of the sender from which the message is to be received are specified as parameters to this call. The *recvfrom* call collects the first message in the queue at the socket. However, if the queue is empty, it blocks until a message arrives.

Figure 3.19 illustrates the use of sockets for connectionless communication between two processes. In the *socket* call, the specification of *AF_INET* as the first parameter indicates that the communication domain is the Internet communication domain, and the specification of *SOCK_DGRAM* as the second parameter indicates that the socket is of the datagram type (used for unreliable, connectionless communication).

Alternatively, Figure 3.20 illustrates the use of sockets for connection-based communication between a client process and a server process. The specification of *SOCK_STREAM* as the second parameter of the *socket* call indicates that the socket is of the stream type (used for reliable, connection-based communication).

3.12 SUMMARY

Interprocess communication (IPC) requires information sharing among two or more processes. The two basic methods for information sharing are original sharing (shared-data approach) and copy sharing (message-passing approach). Since computers in a network do not share memory, the message-passing approach is most commonly used in distributed systems.

A message-passing system is a subsystem of a distributed operating system that provides a set of message-based protocols, and it does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers.

CHAPTER 4

Remote Procedure Calls

4.1 INTRODUCTION

The general message-passing model of interprocess communication (IPC) was presented in the previous chapter. The IPC part of a distributed application can often be adequately and efficiently handled by using an IPC protocol based on the message-passing model. However, an independently developed IPC protocol is tailored specifically to one application and does not provide a foundation on which to build a variety of distributed applications. Therefore, a need was felt for a general IPC protocol that can be used for designing several distributed applications. The Remote Procedure Call (RPC) facility emerged out of this need. It is a special case of the general message-passing model of IPC. Providing the programmers with a familiar mechanism for building distributed systems is one of the primary motivations for developing the RPC facility. While the RPC facility is not a universal panacea for all types of distributed applications, it does provide a valuable communication mechanism that is suitable for building a fairly large number of distributed applications.

The RPC has become a widely accepted IPC mechanism in distributed systems. The popularity of RPC as the primary communication mechanism for distributed applications is due to its following features:

1. Simple call syntax.
2. Familiar semantics (because of its similarity to local procedure calls).
3. Its specification of a well-defined interface. This property is used to support compile-time type checking and automated interface generation.
4. Its ease of use. The clean and simple semantics of a procedure call makes it easier to build distributed computations and to get them right.
5. Its generality. This feature is owing to the fact that in single-machine computations procedure calls are often the most important mechanism for communication between parts of the algorithm [Birrell and Nelson 1984].
6. Its efficiency. Procedure calls are simple enough for communication to be quite rapid.
7. It can be used as an IPC mechanism to communicate between processes on different machines as well as between different processes on the same machine.

4.2 THE RPC MODEL

The RPC model is similar to the well-known and well-understood procedure call model used for the transfer of control and data within a program in the following manner:

1. For making a procedure call, the caller places arguments to the procedure in some well-specified location.
2. Control is then transferred to the sequence of instructions that constitutes the body of the procedure.
3. The procedure body is executed in a newly created execution environment that includes copies of the arguments given in the calling instruction.
4. After the procedure's execution is over, control returns to the calling point, possibly returning a result.

The RPC mechanism is an extension of the procedure call mechanism in the sense that it enables a call to be made to a procedure that does not reside in the address space of the calling process. The called procedure (commonly called *remote procedure*) may be on the same computer as the calling process or on a different computer.

In case of RPC, since the caller and the callee processes have disjoint address spaces (possibly on different computers), the remote procedure has no access to data and variables of the caller's environment. Therefore the RPC facility uses a message-passing scheme for information exchange between the caller and the callee processes. As shown in Figure 4.1, when a remote procedure call is made, the caller and the callee processes interact in the following manner:

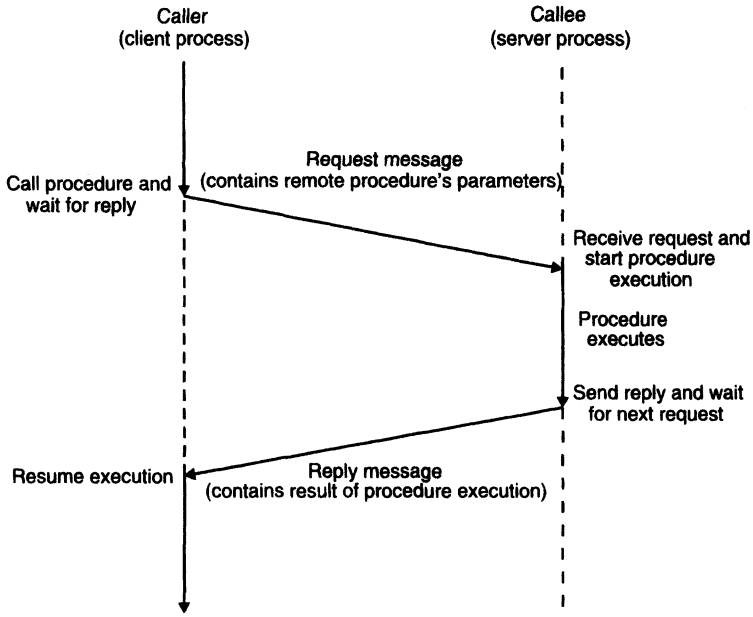


Fig. 4.1 A typical model of Remote Procedure Call.

1. The caller (commonly known as *client process*) sends a call (request) message to the callee (commonly known as *server process*) and waits (blocks) for a reply message. The request message contains the remote procedure's parameters, among other things.
2. The server process executes the procedure and then returns the result of procedure execution in a reply message to the client process.
3. Once the reply message is received, the result of procedure execution is extracted, and the caller's execution is resumed.

The server process is normally dormant, awaiting the arrival of a request message. When one arrives, the server process extracts the procedure's parameters, computes the result, sends a reply message, and then awaits the next call message.

Note that in this model of RPC, only one of the two processes is active at any given time. However, in general, the RPC protocol makes no restrictions on the concurrency model implemented, and other models of RPC are possible depending on the details of the parallelism of the caller's and callee's environments and the RPC implementation. For example, an implementation may choose to have RPC calls to be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a thread (threads are described in Chapter 8) to process an incoming request, so that the server can be free to receive other requests.

4.3 TRANSPARENCY OF RPC

A major issue in the design of an RPC facility is its transparency property. A transparent RPC mechanism is one in which local procedures and remote procedures are (effectively) indistinguishable to programmers. This requires the following two types of transparencies [Wilbur and Bacarisse 1987]:

1. *Syntactic transparency* means that a remote procedure call should have exactly the same syntax as a local procedure call.
2. *Semantic transparency* means that the semantics of a remote procedure call are identical to those of a local procedure call.

It is not very difficult to achieve syntactic transparency of an RPC mechanism, and we have seen that the semantics of remote procedure calls are also analogous to that of local procedure calls for most parts:

- The calling process is suspended until the called procedure returns.
- The caller can pass arguments to the called procedure (remote procedure).
- The called procedure (remote procedure) can return results to the caller.

Unfortunately, achieving exactly the same semantics for remote procedure calls as for local procedure calls is close to impossible [Tanenbaum and Van Renesse 1988]. This is mainly because of the following differences between remote procedure calls and local procedure calls:

1. Unlike local procedure calls, with remote procedure calls, the called procedure is executed in an address space that is disjoint from the calling program's address space. Due to this reason, the called (remote) procedure cannot have access to any variables or data values in the calling program's environment. Thus in the absence of shared memory, it is meaningless to pass addresses in arguments, making call-by-reference pointers highly unattractive. Similarly, it is meaningless to pass argument values containing pointer structures (e.g., linked lists), since pointers are normally represented by memory addresses. According to Bal et al. [1989], dereferencing a pointer passed by the caller has to be done at the caller's side, which implies extra communication. An alternative implementation is to send a copy of the value pointed at the receiver, but this has subtly different semantics and may be difficult to implement if the pointer points into the middle of a complex data structure, such as a directed graph. Similarly, call by reference can be replaced by copy in/copy out, but at the cost of slightly different semantics.

2. Remote procedure calls are more vulnerable to failure than local procedure calls, since they involve two different processes and possibly a network and two different computers. Therefore programs that make use of remote procedure calls must have the capability of handling even those errors that cannot occur in local procedure calls. The need for the ability to take care of the possibility of processor crashes and communication

problems of a network makes it even more difficult to obtain the same semantics for remote procedure calls as for local procedure calls.

3. Remote procedure calls consume much more time (100–1000 times more) than local procedure calls. This is mainly due to the involvement of a communication network in RPCs. Therefore applications using RPCs must also have the capability to handle the long delays that may possibly occur due to network congestion.

Because of these difficulties in achieving normal call semantics for remote procedure calls, some researchers feel that the RPC facility should be nontransparent. For example, Hamilton [1984] argues that remote procedures should be treated differently from local procedures from the start, resulting in a nontransparent RPC mechanism. Similarly, the designers of RPC in Argus [Liskov and Scheifler 1983] were of the opinion that although the RPC system should hide low-level details of message passing from the users, failures and long delays should not be hidden from the caller. That is, the caller should have the flexibility of handling failures and long delays in an application-dependent manner. In conclusion, although in most environments total semantic transparency is impossible, enough can be done to ensure that distributed application programmers feel comfortable.

4.4 IMPLEMENTING RPC MECHANISM

To achieve the goal of semantic transparency, the implementation of an RPC mechanism is based on the concept of *stubs*, which provide a perfectly normal (local) procedure call abstraction by concealing from programs the interface to the underlying RPC system. We saw that an RPC involves a client process and a server process. Therefore, to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes. Moreover, to hide the existence and functional details of the underlying network, an RPC communication package (known as *RPCRuntime*) is used on both the client and server sides. Thus, implementation of an RPC mechanism usually involves the following five elements of program [Birrell and Nelson 1984]:

1. The client
2. The client stub
3. The RPCRuntime
4. The server stub
5. The server

The interaction between them is shown in Figure 4.2. The client, the client stub, and one instance of RPCRuntime execute on the client machine, while the server, the server stub, and another instance of RPCRuntime execute on the server machine. The job of each of these elements is described below.

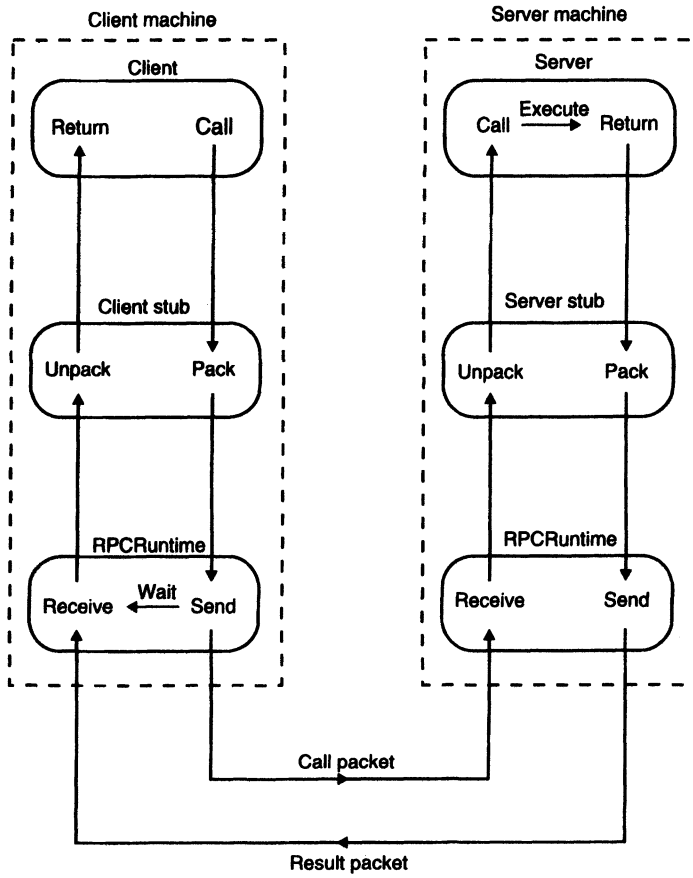


Fig. 4.2 Implementation of RPC mechanism.

4.4.1 Client

The client is a user process that initiates a remote procedure call. To make a remote procedure call, the client makes a perfectly normal local call that invokes a corresponding procedure in the client stub.

4.4.2 Client Stub

The client stub is responsible for carrying out the following two tasks:

- On receipt of a call request from the client, it packs a specification of the target procedure and the arguments into a message and then asks the local RPCRuntime to send it to the server stub.

- On receipt of the result of procedure execution, it unpacks the result and passes it to the client.

4.4.3 RPCRuntime

The RPCRuntime handles transmission of messages across the network between client and server machines. It is responsible for retransmissions, acknowledgments, packet routing, and encryption. The RPCRuntime on the client machine receives the call request message from the client stub and sends it to the server machine. It also receives the message containing the result of procedure execution from the server machine and passes it to the client stub.

On the other hand, the RPCRuntime on the server machine receives the message containing the result of procedure execution from the server stub and sends it to the client machine. It also receives the call request message from the client machine and passes it to the server stub.

4.4.4 Server Stub

The job of the server stub is very similar to that of the client stub. It performs the following two tasks:

- On receipt of the call request message from the local RPCRuntime, the server stub unpacks it and makes a perfectly normal call to invoke the appropriate procedure in the server.
- On receipt of the result of procedure execution from the server, the server stub packs the result into a message and then asks the local RPCRuntime to send it to the client stub.

4.4.5 Server

On receiving a call request from the server stub, the server executes the appropriate procedure and returns the result of procedure execution to the server stub.

Note here that the beauty of the whole scheme is the total ignorance on the part of the client that the work was done remotely instead of by the local kernel. When the client gets control following the procedure call that it made, all it knows is that the results of the procedure execution are available to it. Therefore, as far as the client is concerned, remote services are accessed by making ordinary (local) procedure calls, not by using the *send* and *receive* primitives of Chapter 3. All the details of the message passing are hidden in the client and server stubs, making the steps involved in message passing invisible to both the client and the server.

4.5 STUB GENERATION

Stubs can be generated in one of the following two ways:

1. *Manually.* In this method, the RPC implementor provides a set of translation functions from which a user can construct his or her own stubs. This method is simple to implement and can handle very complex parameter types.

2. *Automatically.* This is the more commonly used method for stub generation. It uses *Interface Definition Language* (*IDL*) that is used to define the interface between a client and a server. An interface definition is mainly a list of procedure names supported by the interface, together with the types of their arguments and results. This is sufficient information for the client and server to independently perform compile-time type-checking and to generate appropriate calling sequences. However, an interface definition also contains other information that helps RPC reduce data storage and the amount of data transferred over the network. For example, an interface definition has information to indicate whether each argument is input, output, or both—only input arguments need be copied from client to server and only output arguments need be copied from server to client. Similarly, an interface definition also has information about type definitions, enumerated types, and defined constants that each side uses to manipulate data from RPC calls, making it unnecessary for both the client and the server to store this information separately. (See Figure 4.21 for an example of an interface definition.)

A server program that implements procedures in an interface is said to *export* the interface, and a client program that calls procedures from an interface is said to *import* the interface. When writing a distributed application, a programmer first writes an interface definition using the IDL. He or she can then write the client program that imports the interface and the server program that exports the interface. The interface definition is processed using an IDL compiler to generate components that can be combined with client and server programs, without making any changes to the existing compilers. In particular, from an interface definition, an IDL compiler generates a client stub procedure and a server stub procedure for each procedure in the interface, the appropriate marshaling and unmarshaling operations (described later in this chapter) in each stub procedure, and a header file that supports the data types in the interface definition. The header file is included in the source files of both the client and server programs, the client stub procedures are compiled and linked with the client program, and the server stub procedures are compiled and linked with the server program. An IDL compiler can be designed to process interface definitions for use with different languages, enabling clients and servers written in different languages, to communicate by using remote procedure calls.

4.6 RPC MESSAGES

Any remote procedure call involves a client process and a server process that are possibly located on different computers. The mode of interaction between the client and server is that the client asks the server to execute a remote procedure and the server returns the

result of execution of the concerned procedure to the client. Based on this mode of interaction, the two types of messages involved in the implementation of an RPC system are as follows:

1. *Call messages* that are sent by the client to the server for requesting execution of a particular remote procedure
2. *Reply messages* that are sent by the server to the client for returning the result of remote procedure execution

The protocol of the concerned RPC system defines the format of these two types of messages. Normally, an RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. Therefore an RPC protocol deals only with the specification and interpretation of these two types of messages.

4.6.1 Call Messages

Since a call message is used to request execution of a particular remote procedure, the two basic components necessary in a call message are as follows:

1. The identification information of the remote procedure to be executed
2. The arguments necessary for the execution of the procedure

In addition to these two fields, a call message normally has the following fields:

3. A message identification field that consists of a sequence number. This field is useful in two ways—for identifying lost messages and duplicate messages in case of system failures and for properly matching reply messages to outstanding call messages, especially in those cases where the replies of several outstanding call messages arrive out of order.
4. A message type field that is used to distinguish call messages from reply messages. For example, in an RPC system, this field may be set to 0 for all call messages and set to 1 for all reply messages.
5. A client identification field that may be used for two purposes—to allow the server of the RPC to identify the client to whom the reply message has to be returned and to allow the server to check the authentication of the client process for executing the concerned procedure.

Thus, a typical RPC call message format may be of the form shown in Figure 4.3.

4.6.2 Reply Messages

When the server of an RPC receives a call message from a client, it could be faced with one of the following conditions. In the list below, it is assumed for a particular condition that no problem was detected by the server for any of the previously listed conditions:

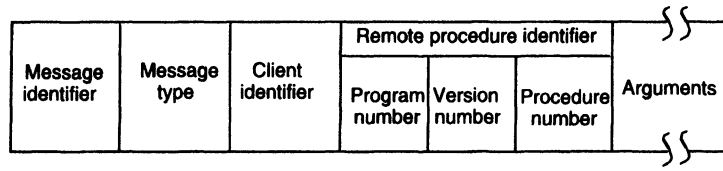


Fig. 4.3 A typical RPC call message format.

1. The server finds that the call message is not intelligible to it. This may happen when a call message violates the RPC protocol. Obviously the server will reject such calls.
2. The server detects by scanning the client's identifier field that the client is not authorized to use the service. The server will return an unsuccessful reply without bothering to make an attempt to execute the procedure.
3. The server finds that the remote program, version, or procedure number specified in the remote procedure identifier field of the call message is not available with it. Again the server will return an unsuccessful reply without bothering to make an attempt to execute the procedure.
4. If this stage is reached, an attempt will be made to execute the remote procedure specified in the call message. Therefore it may happen that the remote procedure is not able to decode the supplied arguments. This may happen due to an incompatible RPC interface being used by the client and server.
5. An exception condition (such as division by zero) occurs while executing the specified remote procedure.
6. The specified remote procedure is executed successfully.

Obviously, in the first five cases, an unsuccessful reply has to be sent to the client with the reason for failure in processing the request and a successful reply has to be sent in the sixth case with the result of procedure execution. Therefore the format of a successful reply message and an unsuccessful reply message is normally slightly different. A typical RPC reply message format for successful and unsuccessful replies may be of the form shown in Figure 4.4.

The message identifier field of a reply message is the same as that of its corresponding call message so that a reply message can be properly matched with its call message. The message type field is properly set to indicate that it is a reply message. For a successful reply, the reply status field is normally set to zero and is followed by the field containing the result of procedure execution. For an unsuccessful reply, the reply status field is either set to 1 or to a nonzero value to indicate failure. In the latter case, the value of the reply status field indicates the type of error. However, in either case, normally a short statement describing the reason for failure is placed in a separate field following the reply status field.

Since RPC protocols are generally independent of transport protocols, it is not possible for an RPC protocol designer to fix the maximum length of call and reply

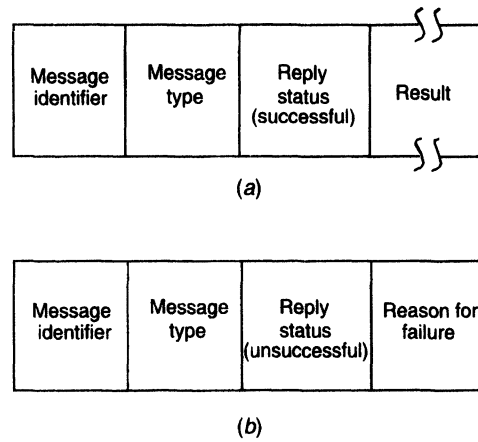


Fig. 4.4 A typical RPC reply message format: (*a*) a successful reply message format; (*b*) an unsuccessful reply message format.

messages. Therefore, for a distributed application to work for a group of transports, it is important for the distributed application developers to ensure that their RPC call and reply messages do not exceed the maximum length specified by any of the transports of the concerned group.

4.7 MARSHALING ARGUMENTS AND RESULTS

Implementation of remote procedure calls involves the transfer of arguments from the client process to the server process and the transfer of results from the server process to the client process. These arguments and results are basically language-level data structures (program objects), which are transferred in the form of message data between the two computers involved in the call. We have seen in the previous chapter that transfer of message data between two computers requires encoding and decoding of the message data. For RPCs this operation is known as *marshaling* and basically involves the following actions:

1. Taking the arguments (of a client process) or the result (of a server process) that will form the message data to be sent to the remote process.
2. Encoding the message data of step 1 above on the sender's computer. This encoding process involves the conversion of program objects into a stream form that is suitable for transmission and placing them into a message buffer.
3. Decoding of the message data on the receiver's computer. This decoding process involves the reconstruction of program objects from the message data that was received in stream form.

In order that encoding and decoding of an RPC message can be performed successfully, the order and the representation method (tagged or untagged) used to

marshal arguments and results must be known to both the client and the server of the RPC. This provides a degree of type safety between a client and a server because the server will not accept a call from a client until the client uses the same interface definition as the server. Type safety is of particular importance to servers since it allows them to survive against corrupt call requests.

The marshaling process must reflect the structure of all types of program objects used in the concerned language. These include primitive types, structured types, and user-defined types. Marshaling procedures may be classified into two groups:

1. Those provided as a part of the RPC software. Normally marshaling procedures for scalar data types, together with procedures to marshal compound types built from the scalar ones, fall in this group.

2. Those that are defined by the users of the RPC system. This group contains marshaling procedures for user-defined data types and data types that include pointers. For example, in Concurrent CLU, developed for use in the Cambridge Distributed Computer System [Bacon and Hamilton 1987], for user-defined types, the type definition must contain procedures for marshaling.

A good RPC system should always generate in-line marshaling code for every remote call so that the users are relieved of the burden of writing their own marshaling procedures. However, practically it is difficult to achieve this goal because of the unacceptable large amounts of code that may have to be generated for handling all possible data types.

4.8 SERVER MANAGEMENT

In RPC-based applications, two important issues that need to be considered for server management are server implementation and server creation.

4.8.1 Server Implementation

Based on the style of implementation used, servers may be of two types: stateful and stateless.

Stateful Servers

A stateful server maintains clients' state information from one remote procedure call to the next. That is, in case of two subsequent calls by a client to a stateful server, some state information pertaining to the service performed for the client as a result of the first call execution is stored by the server process. These clients' state information is subsequently used at the time of executing the second call.

For example, let us consider a server for byte-stream files that allows the following operations on files:

Open (*filename, mode*): This operation is used to open a file identified by *filename* in the specified *mode*. When the server executes this operation, it creates an entry for this file in a *file-table* that it uses for maintaining the file state information of all the open files. The file state information normally consists of the identifier of the file, the open mode, and the current position of a nonnegative integer pointer, called the *read-write pointer*. When a file is opened, its *read-write pointer* is set to zero and the server returns to the client a file identifier (*fid*), which is used by the client for subsequent accesses to that file.

Read (*fid, n, buffer*): This operation is used to get *n* bytes of data from the file identified by *fid* into the buffer named *buffer*. When the server executes this operation, it returns to the client *n* bytes of file data starting from the byte currently addressed by the *read-write pointer* and then increments the *read-write pointer* by *n*.

Write (*fid, n, buffer*): On execution of this operation, the server takes *n* bytes of data from the specified *buffer*, writes it into the file identified by *fid* at the byte position currently addressed by the *read-write pointer*, and then increments the *read-write pointer* by *n*.

Seek (*fid, position*): This operation causes the server to change the value of the *read-write pointer* of the file identified by *fid* to the new value specified as *position*.

Close (*fid*): This statement causes the server to delete from its *file-table* the file state information of the file identified by *fid*.

The file server mentioned above is stateful because it maintains the current state information for a file that has been opened for use by a client. Therefore, as shown in Figure 4.5, after opening a file, if a client makes two subsequent **Read** (*fid, 100, buf*) calls, the first call will return the first 100 bytes (bytes 0–99) and the second call will return the next 100 bytes (bytes 100–199).

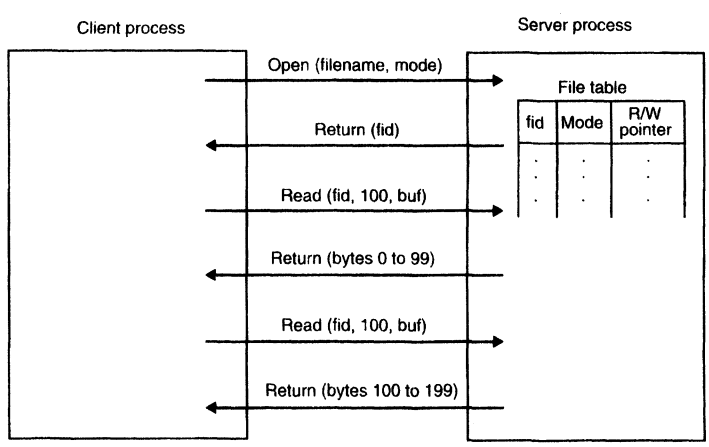


Fig. 4.5 An example of a stateful file server.

Stateless Servers

A stateless server does not maintain any client state information. Therefore every request from a client must be accompanied with all the necessary parameters to successfully carry out the desired operation. For example, a server for byte stream files that allows the following operations on files is stateless.

Read (*filename, position, n, buffer*): On execution of this operation, the server returns to the client *n* bytes of data of the file identified by *filename*. The returned data is placed in the buffer named *buffer*. The value of actual number of bytes read is also returned to the client. The position within the file from where to begin reading is specified as the *position* parameter.

Write (*filename, position, n, buffer*): When the server executes this operation, it takes *n* bytes of data from the specified *buffer* and writes it into the file identified by *filename*. The *position* parameter specifies the byte position within the file from where to start writing. The server returns to the client the actual number of bytes written.

As shown in Figure 4.6, this file server does not keep track of any file state information resulting from a previous operation. Therefore if a client wishes to have similar effect as that in Figure 4.5, the following two *Read* operations must be carried out:

Read (*filename, 0, 100, buf*)
Read (*filename, 100, 100, buf*)

Notice that in this case the client has to keep track of the file state information.

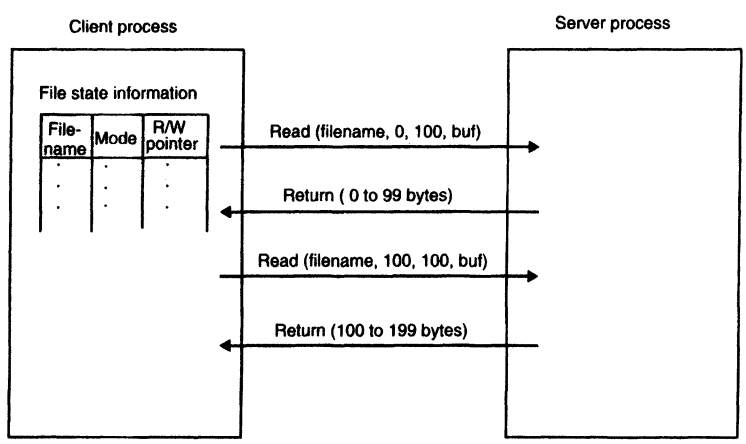


Fig. 4.6 An example of a stateless file server.

Why Stateless Servers?

From the description of stateful and stateless servers, readers might have observed that stateful servers provide an easier programming paradigm because they relieve the clients from the task of keeping track of state information. In addition, stateful servers are typically more efficient than stateless servers. Therefore, the obvious question that arises is why should stateless servers be used at all.

The use of stateless servers in many distributed applications is justified by the fact that stateless servers have a distinct advantage over stateful servers in the event of a failure. For example, with stateful servers, if a server crashes and then restarts, the state information that it was holding may be lost and the client process might continue its task unaware of the crash, producing inconsistent results. Similarly, when a client process crashes and then restarts its task, the server is left holding state information that is no longer valid but cannot easily be withdrawn. Therefore, the client of a stateful server must be properly designed to detect server crashes so that it can perform necessary error-handling activities. On the other hand, with stateless servers, a client has to only retry a request until the server responds; it does not need to know that the server has crashed or that the network temporarily went down. Therefore, stateless servers, which can be constructed around repeatable operations, make crash recovery very easy.

Both stateless and stateful servers have their own advantages and disadvantages. The choice of using a stateless or a stateful server is purely application dependent. Therefore, distributed application system designers must carefully examine the positive and negative aspects of both approaches for their applications before making a choice.

4.8.2 Server Creation Semantics

In RPC, the remote procedure to be executed as a result of a remote procedure call made by a client process lies in a server process that is totally independent of the client process. Independence here means that the client and server processes have separate lifetimes, they normally run on separate machines, and they have their own address spaces. Since a server process is independent of a client process that makes a remote procedure call to it, server processes may either be created and installed before their client processes or be created on a demand basis. Based on the time duration for which RPC servers survive, they may be classified as instance-per-call servers, instance-per-transaction/session servers, or persistent servers.

Instance-per-Call Servers

Servers belonging to this category exist only for the duration of a single call. A server of this type is created by RPCRuntime on the server machine only when a call message arrives. The server is deleted after the call has been executed.

This approach for server creation is not commonly used because of the following problems associated with it:

- The servers of this type are stateless because they are killed as soon as they have serviced the call for which they were created. Therefore, any state that has to be

preserved across server calls must be taken care of by either the client process or the supporting operating system. The involvement of the operating system in maintaining intercall state information will make the remote procedure calls expensive. On the other hand, if the intercall state information is maintained by the client process, the state information must be passed to and from the server with each call. This will lead to the loss of data abstraction across the client-server interface, which will ultimately result in loss of attractiveness of the RPC mechanism to the programmers.

- When a distributed application needs to successively invoke the same type of server several times, this approach appears more expensive, since resource (memory space to provide buffer space and control structures) allocation and deallocation has to be done many times. Therefore, the overhead involved in server creation and destruction dominates the cost of remote procedure calls.

Instance-per-Session Servers

Servers belonging to this category exist for the entire session for which a client and a server interact. Since a server of this type exists for the entire session, it can maintain intercall state information, and the overhead involved in server creation and destruction for a client-server session that involves a large number of calls is also minimized.

In this method, normally there is a server manager for each type of service. All these server managers are registered with the binding agent (binding agent mechanism for binding a client and a server is described later in this chapter). When a client contacts the binding agent, it specifies the type of service needed and the binding agent returns the address of the server manager of the desired type to the client. The client then contacts the concerned server manager, requesting it to create a server for it. The server manager then spawns a new server and passes back its address to the client. The client now directly interacts with this server for the entire session. This server is exclusively used by the client for which it was created and is destroyed when the client informs back to the server manager of the corresponding type that it no longer needs that server.

A server of this type can retain useful state information between calls and so can present a cleaner, more abstract interface to its clients. Note that a server of this type only services a single client and hence only has to manage a single set of state information.

Persistent Servers

A persistent server generally remains in existence indefinitely. Moreover, we saw that the servers of the previous two types cannot be shared by two or more clients because they are exclusively created for a particular client on demand. Unlike them, a persistent server is usually shared by many clients.

Servers of this type are usually created and installed before the clients that use them. Each server independently exports its service by registering itself with the binding agent. When a client contacts the binding agent for a particular type of service, the binding agent selects a server of that type either arbitrarily or based on some in-built policy (such as the

minimum number of clients currently bound to it) and returns the address of the selected server to the client. The client then directly interacts with that server.

Note that a persistent server may be simultaneously bound to several clients. In this case, the server interleaves requests from a number of clients and thus has to concurrently manage several sets of state information. If a persistent server is shared by multiple clients, the remote procedure that it offers must be designed so that interleaved or concurrent requests from different clients do not interfere with each other.

Persistent servers may also be used for improving the overall performance and reliability of the system. For this, several persistent servers that provide the same type of service may be installed on different machines to provide either load balancing or some measure of resilience to failure.

4.9 PARAMETER-PASSING SEMANTICS

The choice of parameter-passing semantics is crucial to the design of an RPC mechanism. The two choices are call-by-value and call-by-reference.

4.9.1 Call-by-Value

In the *call-by-value* method, all parameters are copied into a message that is transmitted from the client to the server through the intervening network. This poses no problems for simple compact types such as integers, counters, small arrays, and so on. However, passing larger data types such as multidimensional arrays, trees, and so on, can consume much time for transmission of data that may not be used. Therefore this method is not suitable for passing parameters involving voluminous data.

An argument in favor of the high cost incurred in passing large parameters by value is that it forces the users to be aware of the expense of remote procedure calls for large-parameter lists. In turn, the users are forced to carefully consider their design of the interface needed between client and server to minimize the passing of unnecessary data. Therefore, before choosing RPC parameter-passing semantics, it is important to carefully review and properly design the client-server interfaces so that parameters become more specific with minimal data being transmitted.

4.9.2 Call-by-Reference

Most RPC mechanisms use the call-by-value semantics for parameter passing because the client and the server exist in different address spaces, possibly even on different types of machines, so that passing pointers or passing parameters *by reference* is meaningless. However, a few RPC mechanisms do allow passing of parameters by reference in which pointers to the parameters are passed from the client to the server. These are usually closed systems, where a single address space is shared by all processes in the system. For example, distributed systems having distributed shared-memory mechanisms (described in Chapter 5) can allow passing of parameters by reference.

In an object-based system that uses the RPC mechanism for object invocation, the call-by-reference semantics is known as *call-by-object-reference*. This is because in an object-based system, the value of a variable is a reference to an object, so it is this reference (the object name) that is passed in an invocation.

Emerald [Black et al. 1986, 1987] designers observed that the use of a call-by-object-reference mechanism in distributed systems presents a potentially serious performance problem because on a remote invocation access by the remote operation to an argument is likely to cause an additional remote invocation. Therefore to avoid many remote references, Emerald supports a new parameter-passing mode that is known as *call-by-move*. In call-by-move, a parameter is passed by reference, as in the method of call-by-object-reference, but at the time of the call, the parameter object is moved to the destination node (site of the callee). Following the call, the argument object may either return to the caller's node or remain at the callee's node (these two modes are known as *call-by-visit* and *call-by-move*, respectively).

Obviously, the use of the call-by-move mode for parameter passing requires that the underlying system supports mobile objects that can be moved from one node to another. Emerald objects are mobile.

Notice that call-by-move does not change the parameter-passing semantics, which is still call-by-object-reference. Therefore call-by-move is basically convenient and optimizes performance. This is because call-by-move could be emulated as a two-step operation:

- First move each call-by-move parameter object to the invokee's node.
- Then invoke the object.

However, performing the moves separately would cause multiple messages to be sent across the network. Thus, providing call-by-move as a parameter-passing mode allows packaging of the argument objects in the same network packet as the invocation message, thereby reducing the network traffic and message count.

Although call-by-move reduces the cost of references made by the invokee, it increases the cost of the invocation itself. If the parameter object is mutable and shared, it also increases the cost of references by the invoker [Black et al. 1987].

4.10 CALL SEMANTICS

In RPC, the caller and the callee processes are possibly located on different nodes. Thus it is possible for either the caller or the callee node to fail independently and later to be restarted. In addition, failure of communication links between the caller and the callee nodes is also possible. Therefore, the normal functioning of an RPC may get disrupted due to one or more of the following reasons:

- The call message gets lost.
- The response message gets lost.

- The callee node crashes and is restarted.
- The caller node crashes and is restarted.

Some element of a caller's node that is involved in the RPC must contain necessary code to handle these failures. Obviously, the code for the caller's procedure should not be forced to deal with these failures. Therefore, the failure-handling code is generally a part of RPCRuntime. The call semantics of an RPC system that determines how often the remote procedure may be executed under fault conditions depends on this part of the RPCRuntime code. This part of the code may be designed to provide the flexibility to the application programmers to select from different possible call semantics supported by an RPC system. The different types of call semantics used in RPC systems are described below.

4.10.1 Possibly or May-Be Call Semantics

This is the weakest semantics and is not really appropriate to RPC but is mentioned here for completeness. In this method, to prevent the caller from waiting indefinitely for a response from the callee, a timeout mechanism is used. That is, the caller waits until a predetermined timeout period and then continues with its execution. Therefore the semantics does not guarantee anything about the receipt of the call message or the procedure execution by the caller. This semantics may be adequate for some applications in which the response message is not important for the caller and where the application operates within a local area network having a high probability of successful transmission of messages.

4.10.2 Last-One Call Semantics

This call semantics is similar to the one described in Section 3.9 and illustrated with an example in Figure 3.10. It uses the idea of retransmitting the call message based on timeouts until a response is received by the caller. That is, the calling of the remote procedure by the caller, the execution of the procedure by the callee, and the return of the result to the caller will eventually be repeated until the result of procedure execution is received by the caller. Clearly, the results of the last executed call are used by the caller, although earlier (abandoned) calls may have had side effects that survived the crash. Hence this semantics is called last-one semantics.

Last-one semantics can be easily achieved in the way described above when only two processors (nodes) are involved in the RPC. However, achieving last-one semantics in the presence of crashes turns out to be tricky for nested RPCs that involve more than two processors (nodes) [Bal et al. 1989]. For example, suppose process $P1$ of node $N1$ calls procedure $F1$ on node $N2$, which in turn calls procedure $F2$ on node $N3$. While the process on $N3$ is working on $F2$, node $N1$ crashes. Node $N1$'s processes will be restarted, and $P1$'s call to $F1$ will be repeated. The second invocation of $F1$ will again call procedure $F2$ on node $N3$. Unfortunately, node $N3$ is totally unaware of node $N1$'s crash. Therefore procedure $F2$ will be executed twice on node $N3$ and $N3$ may return the results of the two executions of $F2$ in any order, possibly violating last-one semantics.

The basic difficulty in achieving last-one semantics in such cases is caused by orphan calls. An *orphan call* is one whose parent (caller) has expired due to a node crash. To achieve last-one semantics, these orphan calls must be terminated before restarting the crashed processes. This is normally done either by waiting for them to finish or by tracking them down and killing them ("*orphan extermination*"). As this is not an easy job, other weaker semantics have been proposed for RPC.

4.10.3 Last-of-Many Call Semantics

This is similar to the last-one semantics except that the orphan calls are neglected [Bal et al. 1989]. A simple way to neglect orphan calls is to use call identifiers to uniquely identify each call. When a call is repeated, it is assigned a new call identifier. Each response message has the corresponding call identifier associated with it. A caller accepts a response only if the call identifier associated with it matches with the identifier of the most recently repeated call; otherwise it ignores the response message.

4.10.4 At-Least-Once Call Semantics

This is an even weaker call semantics than the last-of-many call semantics. It just guarantees that the call is executed one or more times but does not specify which results are returned to the caller. It can be implemented simply by using timeout-based retransmissions without caring for the orphan calls. That is, for nested calls, if there are any orphan calls, it takes the result of the first response message and ignores the others, whether or not the accepted response is from an orphan.

4.10.5 Exactly-Once Call Semantics

This is the strongest and the most desirable call semantics because it eliminates the possibility of a procedure being executed more than once no matter how many times a call is retransmitted. The last-one, last-of-many, and at-least-once call semantics cannot guarantee this. The main disadvantage of these cheap semantics is that they force the application programmer to design idempotent interfaces that guarantee that if a procedure is executed more than once with the same parameters, the same results and side effects will be produced. For example, let us consider the example given in [Wilbur and Bacarisse 1987] for reading and writing a record in a sequential file of fixed-length records. For reading successive records from such a file, a suitable procedure is

```
ReadNextRecord(Filename)
```

Ignoring initialization and end-of-file effects, each execution of this procedure will return the next record from the specified file. Obviously, this procedure is not idempotent because multiple execution of this procedure will return the successive records, which is not desirable for duplicate calls that are retransmitted due to the loss of response messages. This happens because in the implementation of this procedure, the server needs

to keep track of the current record position for each client that has opened the file for accessing. Therefore to design an idempotent interface for reading the next record from the file, it is important that each client keeps track of its own current record position and the server is made stateless, that is, no client state should be maintained on the server side. Based on this idea, an idempotent procedure for reading the next record from a sequential file is

```
ReadRecordN(Filename, N)
```

which returns the *N*th record from the specified file. In this case, the client has to correctly specify the value of *N* to get the desired record from the file.

However, not all nonidempotent interfaces can be so easily transformed to an idempotent form. For example, consider the following procedure for appending a new record to the same sequential file:

```
AppendRecord(Filename, Record)
```

It is clearly not idempotent since repeated execution will add further copies of the same record to the file. This interface may be converted into an idempotent interface by using the following two procedures instead of the one defined above:

```
GetLastRecordNo(Filename)
WriteRecordN(Filename, Record, N)
```

The first procedure returns the record number of the last record currently in the file, and the second procedure writes a record at a specified position in the file. Now, for appending a record, the client will have to use the following two procedures:

```
Last = GetLastRecordNo(Filename)
WriteRecordN(Filename, Record, Last)
```

For exactly-once semantics, the programmer is relieved of the burden of implementing the server procedure in an idempotent manner because the call semantics itself takes care of executing the procedure only once. As already described in Section 3.9 and illustrated with an example in Figure 3.12, the implementation of exactly-once call semantics is based on the use of timeouts, retransmissions, call identifiers with the same identifier for repeated calls, and a reply cache associated with the callee.

4.11 COMMUNICATION PROTOCOLS FOR RPCs

Different systems, developed on the basis of remote procedure calls, have different IPC requirements. Based on the needs of different systems, several communication protocols have been proposed for use in RPCs. A brief description of these protocols is given below.

4.11.1 The Request Protocol

This protocol is also known as the *R* (request) protocol [Spector 1982]. It is used in RPCs in which the called procedure has nothing to return as the result of procedure execution and the client requires no confirmation that the procedure has been executed. Since no acknowledgment or reply message is involved in this protocol, only one message per call is transmitted (from client to server) (Fig. 4.7). The client normally proceeds immediately after sending the request message as there is no need to wait for a reply message. The protocol provides may-be call semantics and requires no retransmission of request messages.

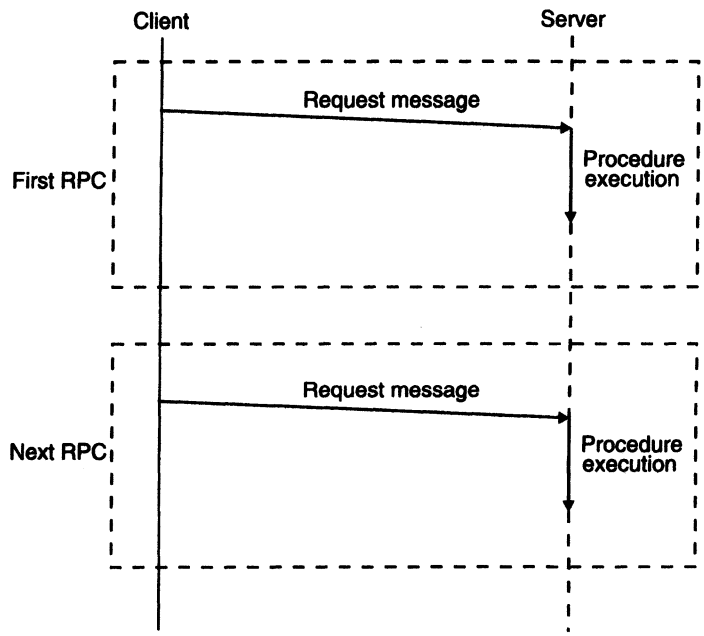


Fig. 4.7 The request (R) protocol.

An RPC that uses the R protocol is called *asynchronous RPC*. An asynchronous RPC helps in improving the combined performance of both the client and the server in those distributed applications in which the client does not need a reply to each request. Client performance is improved because the client is not blocked and can immediately continue to do other work after making the call. On the other hand, server performance is improved because the server need not generate and send any reply for the request. One such application is a distributed window system. A distributed window system, such as X-11 [Davison et al. 1992], is programmed as a server, and application programs wishing to display items in windows on a display screen are its clients. To display items in a window, a client normally sends many requests (each request containing a relatively small amount

of information for a small change in the displayed information) to the server one after another without waiting for a reply for each of these requests because it does not need replies for the requests.

Notice that for an asynchronous RPC, the RPCRuntime does not take responsibility for retrying a request in case of communication failure. This means that if an unreliable datagram transport protocol such as UDP is used for the RPC, the request message could be lost without the client's knowledge. Applications using asynchronous RPC with unreliable transport protocol must be prepared to handle this situation. However, if a reliable, connection-oriented transport protocol such as TCP is used for the RPC, there is no need to worry about retransmitting the request message because it is delivered reliably in this case.

Asynchronous RPCs with unreliable transport protocol are generally useful for implementing periodic update services. For example, a time server node in a distributed system may send time synchronization messages every T seconds to other nodes using the asynchronous RPC facility. In this case, even if a message is lost, the correct time is transmitted in the next message. Each node can keep track of the last time it received an update message to prevent it from missing too many update messages. A node that misses too many update messages can send a special request message to the time server node to get a reliable update after some maximum amount of time.

4.11.2 The Request/Reply Protocol

This protocol is also known as the *RR* (request/reply) protocol [Spector 1982]. It is useful for the design of systems involving simple RPCs. A *simple RPC* is one in which all the arguments as well as all the results fit in a single packet buffer and the duration of a call and the interval between calls are both short (less than the transmission time for a packet between the client and server) [Birrell and Nelson 1984]. The protocol is based on the idea of using implicit acknowledgment to eliminate explicit acknowledgment messages. Therefore in this protocol:

- A server's reply message is regarded as an acknowledgment of the client's request message.
- A subsequent call packet from a client is regarded as an acknowledgment of the server's reply message of the previous call made by that client.

The exchange of messages between a client and a server in the RR protocol is shown in Figure 4.8. Notice from the figure that the protocol involves the transmission of only two packets per call (one in each direction).

The RR protocol in its basic form does not possess failure-handling capabilities. Therefore to take care of lost messages, the timeouts-and-retries technique is normally used along with the RR protocol. In this technique, a client retransmits its request message if it does not receive the response message before a predetermined timeout period elapses. Obviously, if duplicate request messages are not filtered out, the RR protocol, compounded with this technique, provides at-least-once call semantics.

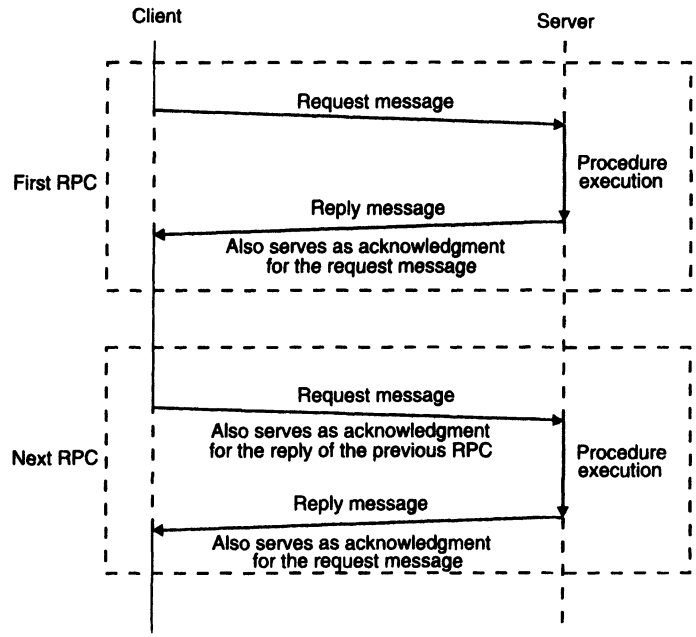


Fig. 4.8 The request/reply (RR) protocol.

However, servers can support exactly-once call semantics by keeping records of the replies in a reply cache that enables them to filter out duplicate request messages and to retransmit reply messages without the need to reprocess a request. The details of this technique were given in Section 3.9.

4.11.3 The Request/Reply/Acknowledge-Reply Protocol

This protocol is also known as the *RRA* (request/reply/acknowledge-reply) protocol [Spector 1982]. The implementation of exactly-once call semantics with RR protocol requires the server to maintain a record of the replies in its reply cache. In situations where a server has a large number of clients, this may result in servers needing to store large quantities of information. In some implementations, servers restrict the quantity of such data by discarding it after a limited period of time. However, this approach is not fully reliable because sometimes it may lead to the loss of those replies that have not yet been successfully delivered to their clients. To overcome this limitation of the RR protocol, the RRA protocol is used, which requires clients to acknowledge the receipt of reply messages. The server deletes an information from its reply cache only after receiving an acknowledgment for it from the client. As shown in Figure 4.9, the RRA protocol involves the transmission of three messages per call (two from the client to the server and one from the server to the client).

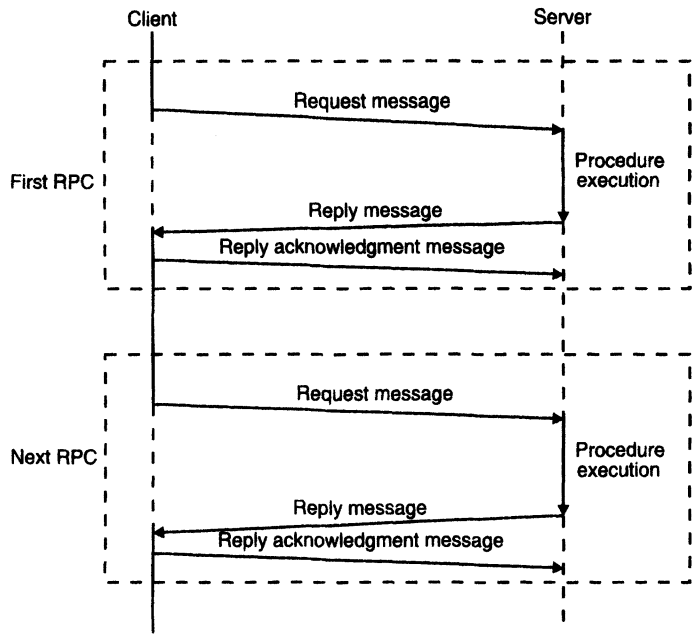


Fig. 4.9 The request/reply/acknowledge-reply (RRA) protocol.

In the RRA protocol, there is a possibility that the acknowledgment message may itself get lost. Therefore implementation of the RRA protocol requires that the unique message identifiers associated with request messages must be ordered. Each reply message contains the message identifier of the corresponding request message, and each acknowledgment message also contains the same message identifier. This helps in matching a reply with its corresponding request and an acknowledgment with its corresponding reply. A client acknowledges a reply message only if it has received the replies to all the requests previous to the request corresponding to this reply. Thus an acknowledgment message is interpreted as acknowledging the receipt of all reply messages corresponding to the request messages with lower message identifiers. Therefore the loss of an acknowledgment message is harmless.

4.12 COMPLICATED RPCs

Birrell and Nelson [1984] categorized the following two types of RPCs as complicated:

1. RPCs involving long-duration calls or large gaps between calls
2. RPCs involving arguments and/or results that are too large to fit in a single-datagram packet

Different protocols are used for handling these two types of complicated RPCs.

4.12.1 RPCs Involving Long-Duration Calls or Large Gaps between Calls

One of the following two methods may be used to handle complicated RPCs that belong to this category [Birrell and Nelson 1984]:

1. *Periodic probing of the server by the client.* In this method, after a client sends a request message to a server, it periodically sends a probe packet to the server, which the server is expected to acknowledge. This allows the client to detect a server's crash or communication link failures and to notify the corresponding user of an exception condition. The message identifier of the original request message is included in each probe packet. Therefore, if the original request is lost, in reply to a probe packet corresponding to that request message, the server intimates the client that the request message corresponding to the probe packet has not been received. Upon receipt of such a reply from the server, the client retransmits the original request.

2. *Periodic generation of an acknowledgment by the server.* In this method, if a server is not able to generate the next packet significantly sooner than the expected retransmission interval, it spontaneously generates an acknowledgment. Therefore for a long-duration call, the server may have to generate several acknowledgments, the number of acknowledgments being directly proportional to the duration of the call. If the client does not receive either the reply for its request or an acknowledgment from the server within a predetermined timeout period, it assumes that either the server has crashed or communication link failure has occurred. In this case, it notifies the concerned user of an exception condition.

4.12.2 RPCs Involving Long Messages

In some RPCs, the arguments and/or results are too large to fit in a single-datagram packet. For example, in a file server, quite large quantities of data may be transferred as input arguments to the *write* operation or as results to the *read* operation. A simple way to handle such an RPC is to use several physical RPCs for one logical RPC. Each physical RPC transfers an amount of data that fits in a single-datagram packet. This solution is inefficient due to a fixed amount of overhead involved with each RPC independent of the amount of data sent.

Another method of handling complicated RPCs of this category is to use multidatagram messages. In this method, a long RPC argument or result is fragmented and transmitted in multiple packets. To improve communication performance, a single acknowledgment packet is used for all the packets of a multidatagram message. In this case, the same approach that was described in Section 3.9 is used to keep track of lost and out-of-sequence packets of a multidatagram RPC message.

Some RPC systems are limited to small sizes. For example, the Sun Microsystem's RPC is limited to 8 kilobytes. Therefore, in these systems, an RPC involving messages larger than the allowed limit must be handled by breaking it up into several physical RPCs.

4.13 CLIENT-SERVER BINDING

It is necessary for a client (actually a client stub) to know the location of a server before a remote procedure call can take place between them. The process by which a client becomes associated with a server so that calls can take place is known as *binding*. From the application level's point of view, the model of binding is that servers "export" operations to register their willingness to provide service and clients' "import" operations, asking the RPCRuntime system to locate a server and establish any state that may be needed at each end [Bershad et al. 1987]. The client-server binding process involves proper handling of several issues:

1. How does a client specify a server to which it wants to get bound?
2. How does the binding process locate the specified server?
3. When is it proper to bind a client to a server?
4. Is it possible for a client to change a binding during execution?
5. Can a client be simultaneously bound to multiple servers that provide the same service?

These binding issues are described below.

4.13.1 Server Naming

The specification by a client of a server with which it wants to communicate is primarily a naming issue. For RPC, Birrell and Nelson [1984] proposed the use of interface names for this purpose. An *interface name* has two parts—a *type* and an *instance*. Type specifies the interface itself and instance specifies a server providing the services within that interface. For example, there may be an interface of type *file_server*, and there may be several instances of servers providing file service. When a client is not concerned with which particular server of an interface services its request, it need not specify the instance part of the interface name.

The type part of an interface usually also has a version number field to distinguish between old and new versions of the interface that may have different sets of procedures or the same set of procedures with different parameters. It is inevitable in the course of distributed application programming that an application needs to be updated after a given version has been released. The use of a version number field allows old and new versions of a distributed application to coexist. One would hope that the new version of an interface would eventually replace all the old versions of the interface. However, experience has shown that it is always better to maintain backward compatibility with old versions of the software because someone might still be using one of the old versions.

According to Birrell and Nelson [1984], the interface name semantics are based on an arrangement between the exporter and the importer. Therefore, interface names are created by the users. They are not dictated by the RPC package. The RPC package only dictates the means by which an importer uses the interface name to locate an exporter.

4.13.2 Server Locating

The interface name of a server is its unique identifier. Thus when a client specifies the interface name of a server for making a remote procedure call, the server must be located before the client's request message can be sent to it. This is primarily a locating issue and any locating mechanism (locating mechanisms are described in Chapter 10) can be used for this purpose. The two most commonly used methods are as follows:

1. *Broadcasting.* In this method, a message to locate the desired server is broadcast to all the nodes from the client node. The nodes on which the desired server is located return a response message. Note that the desired server may be replicated on several nodes so the client node will receive a response from all these nodes. Normally, the first response that is received at the client's node is given to the client process and all subsequent responses are discarded.

This method is easy to implement and is suitable for use for small networks. However, the method is expensive for large networks because of the increase in message traffic due to the involvement of all the nodes in broadcast processing. Therefore the second method, which is based on the idea of using a name server, is generally used for large networks.

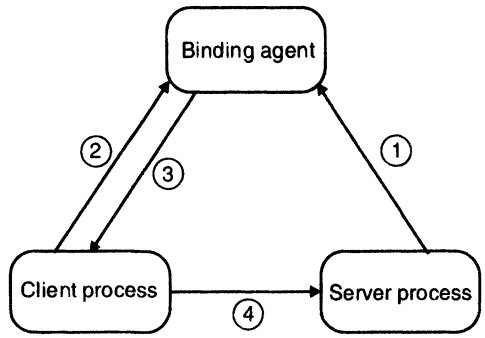
2. *Binding agent.* A binding agent is basically a name server used to bind a client to a server by providing the client with the location information of the desired server. In this method, a binding agent maintains a binding table, which is a mapping of a server's interface name to its locations. All servers register themselves with the binding agent as a part of their initialization process. To register with the binding agent, a server gives the binder its identification information and a handle used to locate it. The handle is system dependent and might be an Ethernet address, an IP address, an X.500 address, a process identifier containing a node number and port number, or something else. A server can also deregister with the binding agent when it is no longer prepared to offer service. The binding agent can also poll the servers periodically, automatically deregistering any server that fails to respond.

To locate a server, a client contacts the binding agent. If the server is registered with the binding agent, it returns the handle (location information) of the server to the client. The method is illustrated in Figure 4.10.

The binding agent's location is known to all nodes. This is accomplished by using either a fixed address for the binding agent that is known to all nodes or a broadcast message to locate the binding agent when a node is booted. In either case, when the binding agent is relocated, a message is sent to all nodes informing the new location of the binding agent.

A binding agent interface usually has three primitives: (a) *register* is used by a server to register itself with the binding agent, (b) *deregister* is used by a server to deregister itself with the binding agent, and (c) *lookup* is used by a client to locate a server.

The binding agent mechanism for locating servers has several advantages. First, the method can support multiple servers having the same interface type so that any of the available servers may be used to service a client's request. This helps to achieve a degree of fault tolerance. Second, since all bindings are done by the binding agent, when multiple



① The server registers itself with the binding agent.

② The client requests the binding agent for the server's location.

③ The binding agent returns the server's location information to the client.

④ The client calls the server.

Fig. 4.10 The binding agent mechanism for locating a server in case of RPC.

servers provide the same service, the clients can be spread evenly over the servers to balance the load. Third, the binding mechanism can be extended to allow servers to specify a list of users who may use its service, in which case the binding agent would refuse to bind those clients to the servers who are not authorized to use its service.

However, the binding agent mechanism has drawbacks. The overhead involved in binding clients to servers is large and becomes significant when many client processes are short lived. Moreover, in addition to any functional requirements, a binding agent must be robust against failures and should not become a performance bottleneck. Distributing the binding function among several binding agents and replicating information among them can satisfy both these criteria. Unfortunately, replication often involves extra overhead of keeping the multiple replicas consistent. Therefore, the functionality offered by many binding agents is lower than might be hoped for.

4.13.3 Binding Time

A client may be bound to a server at compile time, at link time, or at call time [Goscinski 1991].

Binding at Compile Time

In this method, the client and server modules are programmed as if they were intended to be linked together. For example, the server's network address can be compiled into the client code by the programmer and then it can be found by looking up the server's name in a file.

The method is extremely inflexible in the sense that if the server moves or the server is replicated or the interface changes, all client programs using the server will have to be found and recompiled. However, the method is useful in certain limited cases. For example, it may be used in an application whose configuration is expected to remain static for a fairly long time.

Binding at Link Time

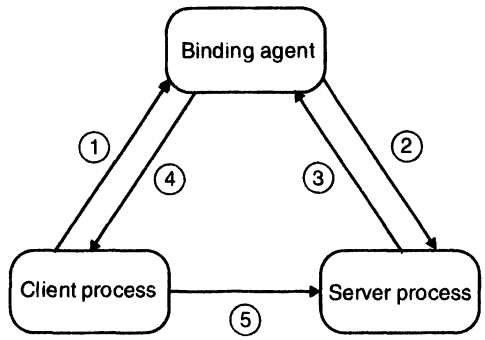
In this method, a server process exports its service by registering itself with the binding agent as part of its initialization process. A client then makes an import request to the binding agent for the service before making a call. The binding agent binds the client and the server by returning to the client the server's handle (details that are necessary for making a call to the server). Calls can take place once the client has received the server's handle. The server's handle is cached by the client to avoid contacting the binding agent for subsequent calls to be made to the same server. Due to the overhead involved in contacting the binding agent, this method is suitable for those situations in which a client calls a server several times once it is bound to it.

Binding at Call Time

In this method, a client is bound to a server at the time when it calls the server for the first time during its execution. A commonly used approach for binding at call time is the *indirect call* method. As shown in Figure 4.11, in this method, when a client calls a server for the first time, it passes the server's interface name and the arguments of the RPC call to the binding agent. The binding agent looks up the location of the target server in its binding table, and on behalf of the client it sends an RPC call message to the target server, including in it the arguments received from the client. When the target server returns the results to the binding agent, the binding agent returns this result to the client along with the target server's handle so that the client can subsequently call the target server directly.

4.13.4 Changing Bindings

The flexibility provided by a system to change bindings dynamically is very useful from a reliability point of view. Binding is a connection establishment between a client and a server. The client or server of a connection may wish to change the binding at some instance of time due to some change in the system state. For example, a client willing to get a request serviced by any one of the multiple servers for that service may be programmed to change a binding to another server of the same type when a call to the already connected server fails. Similarly, the server of a binding may want to alter the binding and connect the client to another server in situations such as when the service needs to move to another node or a new version of the server is installed. When a binding is altered by the concerned server, it is



① The client process passes the server's interface name and the arguments of the RPC call to the binding agent.

② The binding agent sends an RPC call message to the server, including in it the arguments received from the client.

③ The server returns the result of request processing to the binding agent.

④ The binding agent returns this result to the client along with the server's handle.

⑤ Subsequent calls are sent directly from the client process to the server process.

Fig. 4.11 Illustrating binding at call time by the method of indirect call.

important to ensure that any state data held by the server is no longer needed or can be duplicated in the replacement server. For example, when a file server has to be replaced with a new one, either it must be replaced when no files are open or the state of all the open files must be transferred from the old server to the new one as a part of the replacement process.

4.13.5 Multiple Simultaneous Bindings

In a system, a service may be provided by multiple servers. We have seen that, in general, a client is bound to a single server of the several servers of the same type. However, there may be situations when it is advantageous for a client to be bound simultaneously to all or multiple servers of the same type. Logically, a binding of this sort gives rise to multicast communication because when a call is made, all the servers bound to the client for that service will receive and process the call. For example, a client may wish to update multiple copies of a file that is replicated at several nodes. For this, the client can be bound simultaneously to file servers of all those nodes where a replica of the file is located.

4.14 EXCEPTION HANDLING

We saw in Figure 4.4 that when a remote procedure cannot be executed successfully, the server reports an error in the reply message. An RPC also fails when a client cannot contact the server of the RPC. An RPC system must have an effective exception-handling mechanism for reporting such failures to clients. One approach to do this is to define an exception condition for each possible error type and have the corresponding exception raised when an error of that type occurs, causing the exception-handling procedure to be called and automatically executed in the client's environment. This approach can be used with those programming languages that provide language constructs for exception handling. Some such programming languages are ADA, CLU [Liskov et al. 1981], and Modula-3 [Nelson 1991, Harbinson 1992]. In C language, signal handlers can be used for the purpose of exception handling.

However, not every language has an exception-handling mechanism. For example, Pascal does not have such a mechanism. RPC systems designed for use with such languages generally use the method provided in conventional operating systems for exception handling. One such method is to return a well-known value to the process, making a system call to indicate failure and to report the type of error by storing a suitable value in a variable in the environment of the calling program. For example, in UNIX the value −1 is used to indicate failure, and the type of error is reported in the global variable *errno*. In an RPC, a return value indicating an error is used both for errors due to failure to communicate with the server and errors reported in the reply message from the server. The details of the type of error is reported by storing a suitable value in a global variable in the client program. This approach suffers from two main drawbacks. First, it requires the client to test every return value. Second, it is not general enough because a return value used to indicate failure may be a perfectly legal value to be returned by a procedure. For example, if the value −1 is used to indicate failure, this value is also the return value of a procedure call with arguments −5 and 4 to a procedure for getting the sum of two numbers.

4.15 SECURITY

Some implementations of RPC include facilities for client and server authentication as well as for providing encryption-based security for calls. For example, in [Birrell and Nelson 1984], callers are given a guarantee of the identity of the callee, and vice versa, by using the authentication service of Grapevine [Birrell et al. 1982]. For full end-to-end encryption of calls and results, the federal data encryption standard [DES 1977] is used in [Birrell and Nelson 1984]. The encryption techniques provide protection from eavesdropping (and conceal patterns of data) and detect attempts at modification, replay, or creation of calls.

In other implementations of RPC that do not include security facilities, the arguments and results of RPC are readable by anyone monitoring communications between the caller and the callee. Therefore in this case, if security is desired, the user must implement his or her own authentication and data encryption mechanisms. When designing an application, the user should consider the following security issues related with the communication of messages:

- Is the authentication of the server by the client required?
- Is the authentication of the client by the server required when the result is returned?
- Is it all right if the arguments and results of the RPC are accessible to users other than the caller and the callee?

These and other security issues are described in detail in Chapter 11.

4.16 SOME SPECIAL TYPES OF RPCs

4.16.1 Callback RPC

In the usual RPC protocol, the caller and callee processes have a client-server relationship. Unlike this, the callback RPC facilitates a peer-to-peer paradigm among the participating processes. It allows a process to be both a client and a server.

Callback RPC facility is very useful in certain distributed applications. For example, remotely processed interactive applications that need user input from time to time or under special conditions for further processing require this type of facility. As shown in Figure 4.12, in such applications, the client process makes an RPC to the concerned server process, and during procedure execution for the client, the server process makes a callback RPC to the client process. The client process takes necessary action based on the server's request and returns a reply for the callback RPC to the server process. On receiving this reply, the server resumes the execution of the procedure and finally returns the result of the initial call to the client. Note that the server may make several callbacks to the client before returning the result of the initial call to the client process.

The ability for a server to call its client back is very important, and care is needed in the design of RPC protocols to ensure that it is possible. In particular, to provide callback RPC facility, the following are necessary:

- Providing the server with the client's handle
- Making the client process wait for the callback RPC
- Handling callback deadlocks

Commonly used methods to handle these issues are described below.

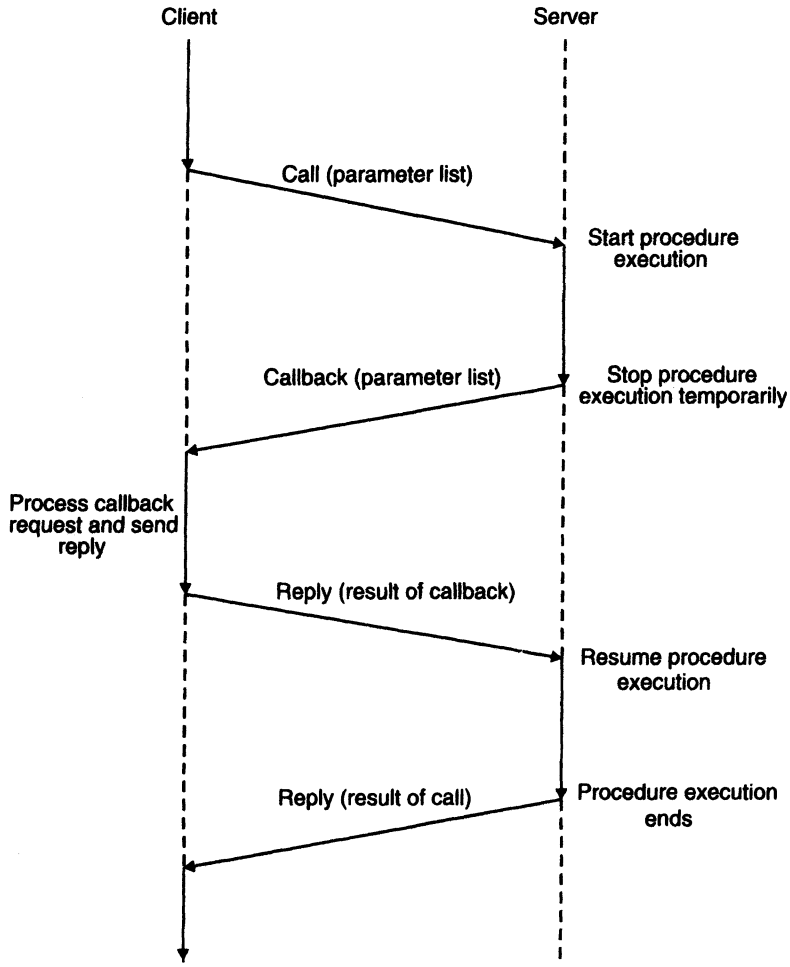


Fig. 4.12 The callback RPC.

Providing the Server with the Client's Handle

The server must have the client's handle to call the client back. The client's handle uniquely identifies the client process and provides enough information to the server for making a call to it. Typically, the client process uses a transient program number for the callback service and exports the callback service by registering its program number with the binding agent. The program number is then sent as a part of the RPC request to the server. To make a callback RPC, the server initiates a normal RPC request to the client using the given program number. Instead of having the client just send the server the program number, it could also send its handle, such as the port number. The client's handle could then be used by the server to directly communicate

with the client and would save an RPC to the binding agent to get the client's handle.

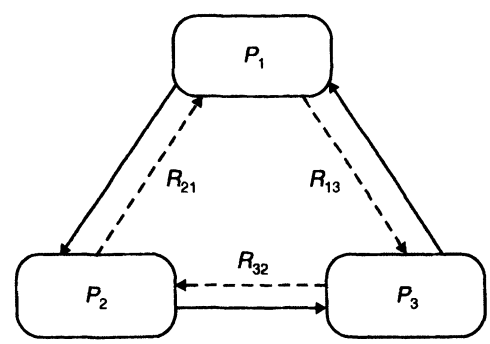
Making the Client Process Wait for the Callback RPC

The client process must be waiting for the callback so that it can process the incoming RPC request from the server and also to ensure that a callback RPC from the server is not mistaken to be the reply of the RPC call made by the client process. To wait for the callback, a client process normally makes a call to a *svc-routine*. The *svc-routine* waits until it receives a request and then dispatches the request to the appropriate procedure.

Handling Callback Deadlocks

In callback RPC, since a process may play the role of either a client or a server, callback deadlocks can occur. For example, consider the most simple case in which a process P_1 makes an RPC call to a process P_2 and waits for a reply from P_2. In the meantime, process P_2 makes an RPC call to another process P_3 and waits for a reply from P_3. In the meantime, process P_3 makes an RPC call to process P_1 and waits for a reply from P_1. But P_1 cannot process P_3's request until its request to P_2 has been satisfied, and P_2 cannot process P_1's request until its request to P_3 has been satisfied, and P_3 cannot process P_2's request until its request to P_1 has been satisfied. As shown in Figure 4.13, a situation now exists where P_1 is waiting for a reply from P_2, which is waiting for a reply from P_3, which is waiting for a reply from P_1. The result is that none of the three processes can have their request satisfied, and hence all three will continue to wait indefinitely. In effect, a callback deadlock has occurred due to the interdependencies of the three processes.

While using a callback RPC, care must be taken to handle callback deadlock situations. Various methods for handling deadlocks are described in Chapter 6.



P_1 is waiting for R_{21} (reply from P_2 to P_1)
P_2 is waiting for R_{32} (reply from P_3 to P_2)
P_3 is waiting for R_{13} (reply from P_1 to P_3)

Fig. 4.13 An example of a callback deadlock in case of callback RPC mechanism.

4.16.2 Broadcast RPC

The RPC-based IPC is normally of the one-to-one type, involving a single client process and a single server process. However, we have seen in the previous chapter that for performance reasons several highly parallel distributed applications require the communication system to provide the facility of broadcast and multicast communication. The RPC-based IPC mechanisms normally support broadcast RPC facility for such applications. In broadcast RPC, a client's request is broadcast on the network and is processed by all the servers that have the concerned procedure for processing that request. The client waits for and receives numerous replies.

A broadcast RPC mechanism may use one of the following two methods for broadcasting a client's request:

1. The client has to use a special broadcast primitive to indicate that the request message has to be broadcasted. The request is sent to the binding agent, which forwards the request to all the servers registered with it. Note that in this method, since all broadcast RPC messages are sent to the binding agent, only services that register themselves with their binding agent are accessible via the broadcast RPC mechanism.

2. The second method is to declare broadcast ports. A network port of each node is connected to a broadcast port. A network port of a node is a queuing point on that node for broadcast messages. The client of the broadcast RPC first obtains a binding for a broadcast port and then broadcasts the RPC message by sending the message to this port. Note that the same primitive may be used for both unicast and broadcast RPCs. Moreover, unlike the first method, this method also has the flexibility of being used for multicast RPC in which the RPC message is sent only to a subset of the available servers. For this, the port declaration mechanism should have the flexibility to associate only a subset of the available servers to a newly declared multicast port.

Since a broadcast RPC message is sent to all the nodes of a network, a reply is expected from each node. As already described in the previous chapter, depending on the degree of reliability desired, the client process may wait for zero, one, *m*-out-of-*n*, or all the replies. In some implementations, servers that support broadcast RPC typically respond only when the request is successfully processed and are silent in the face of errors. Such systems normally use some type of timeout-based retransmission mechanism for improving the reliability of the broadcast RPC protocol. For example, in SunOS, the broadcast RPC protocol transmits the broadcast and waits for 4 seconds before retransmitting the request. It then waits for 6 seconds before retransmitting the request and continues to increment the amount of time to wait by 2 seconds until the timeout period becomes greater than 14 seconds. Therefore, in the worst case, the request is broadcast six times and the total wait time is 54 seconds ($4+6+8+10+12+14$). In SunOS, the broadcast RPC uses unreliable, packet-based protocol for broadcasting the request, and so the routine retransmits the broadcast requests by default. Increasing the amount of time between retransmissions is known as a *back-off algorithm*. The use of a back-off algorithm for timeout-based retransmissions helps in reducing the load on the physical network and computers involved.

4.16.3 Batch-Mode RPC

Batch-mode RPC is used to queue separate RPC requests in a transmission buffer on the client side and then send them over the network in one batch to the server. This helps in the following two ways:

1. It reduces the overhead involved in sending each RPC request independently to the server and waiting for a response for each request.
2. Applications requiring higher call rates (50–100 remote calls per second) may not be feasible with most RPC implementations. Such applications can be accommodated with the use of batch-mode RPC.

However, batch-mode RPC can be used only with those applications in which a client has many RPC requests to send to a server and the client does not need any reply for a sequence of requests. Therefore, the requests are queued on the client side, and the entire queue of requests is flushed to the server when one of the following conditions becomes true:

1. A predetermined interval elapses.
2. A predetermined number of requests have been queued.
3. The amount of batched data exceeds the buffer size.
4. A call is made to one of the server's procedures for which a result is expected. From a programming standpoint, the semantics of such a call (nonqueueing RPC request) should be such that the server can distinguish it from the queued requests and send a reply for it to the client.

The flushing out of queued requests in cases 1, 2, and 3 happens independent of a nonqueuing RPC request and is not noticeable by the client.

Obviously, the queued messages should be sent reliably. Hence, a batch-mode RPC mechanism requires reliable transports such as TCP. Moreover, although the batch-mode optimization retains syntactic transparency, it may produce obscure timing-related effects where other clients are accessing the server simultaneously.

4.17 RPC IN HETEROGENEOUS ENVIRONMENTS

Heterogeneity is an important issue in the design of any distributed application because typically the more portable an application, the better. When designing an RPC system for a heterogeneous environment, the three common types of heterogeneity that need to be considered are as follows:

1. *Data representation.* Machines having different architectures may use different data representations. For example, integers may be represented with the most significant byte at the low-byte address in one machine architecture and at the high-byte address in another machine architecture. Similarly, integers may be represented in 1's complement

notation in one machine architecture and in 2's complement notation in another machine architecture. Floating-point representations may also vary between two different machine architectures. Therefore, an RPC system for a heterogeneous environment must be designed to take care of such differences in data representations between the architectures of client and server machines of a procedure call.

2. *Transport protocol.* For better portability of applications, an RPC system must be independent of the underlying network transport protocol. This will allow distributed applications using the RPC system to be run on different networks that use different transport protocols.

3. *Control protocol.* For better portability of applications, an RPC system must also be independent of the underlying network control protocol that defines control information in each transport packet to track the state of a call.

The most commonly used approach to deal with these types of heterogeneity while designing an RPC system for a heterogeneous environment is to delay the choices of data representation, transport protocol, and control protocol until bind time. In conventional RPC systems, all these decisions are made when the RPC system is designed. That is, the binding mechanism of an RPC system for a heterogeneous environment is considerably richer in information than the binding mechanism used by a conventional RPC system. It includes mechanisms for determining which data conversion software (if any conversion is needed), which transport protocol, and which control protocol should be used between a specific client and server and returns the correct procedures to the stubs as result parameters of the binding call. These binding mechanism details are transparent to the users. That is, application programs never directly access the component structures of the binding mechanism; they deal with bindings only as atomic types and acquire and discard them via the calls of the RPC system.

Some RPC systems designed for heterogeneous environments are the HCS (Heterogeneous Computer Systems) RPC (called HRPC) [Bershad et al. 1987], the DCE SRC (System Research Center) Firefly RPC [Schroeder and Burrows 1990], Matchmaker [Jones et al. 1985], and Horus [Gibbons 1987].

4.18 LIGHTWEIGHT RPC

The *Lightweight Remote Procedure Call* (*LRPC*) was introduced by Bershad et al. [1990] and integrated into the Taos operating system of the DEC SRC Firefly microprocessor workstation. The description below is based on the material in their paper [Bershad et al. 1990].

As mentioned in Chapter 1, based on the size of the kernel, operating systems may be broadly classified into two categories—monolithic-kernel operating systems and microkernel operating systems. Monolithic-kernel operating systems have a large, monolithic kernel that is insulated from user programs by simple hardware boundaries. On the other hand, in microkernel operating systems, a small kernel provides only primitive operations and most of the services are provided by user-level servers. The servers are

usually implemented as processes and can be programmed separately. Each server forms a component of the operating system and usually has its own address space. As compared to the monolithic-kernel approach, in this approach services are provided less efficiently because the various components of the operating system have to use some form of IPC to communicate with each other. The advantages of this approach include simplicity and flexibility. Due to modular structure, microkernel operating systems are simple and easy to design, implement, and maintain.

In the microkernel approach, when different components of the operating system have their own address spaces, the address space of each component is said to form a *domain*, and messages are used for all interdomain communication. In this case, the communication traffic in operating systems are of two types [Bershad et al. 1990]:

1. *Cross-domain*, which involves communication between domains on the same machine
2. *Cross-machine*, which involves communication between domains located on separate machines

The LRPC is a communication facility designed and optimized for cross-domain communications.

Although conventional RPC systems can be used for both cross-domain and cross-machine communications, Bershad et al. observed that the use of conventional RPC systems for cross-domain communications, which dominate cross-machine communications, incurs an unnecessarily high cost. This cost leads system designers to coalesce weakly related components of microkernel operating systems into a single domain, trading safety and performance. Therefore, the basic advantages of using the microkernel approach are not fully exploited. Based on these observations, Bershad et al. designed the LRPC facility for cross-domain communications, which has better performance than conventional RPC systems. Nonetheless, LRPC is safe and transparent and represents a viable communication alternative for microkernel operating systems.

To achieve better performance than conventional RPC systems, the four techniques described below are used by LRPC.

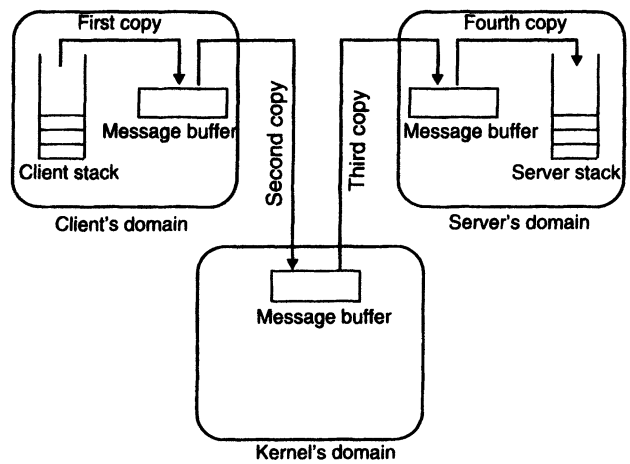
4.18.1 Simple Control Transfer

Whenever possible, LRPC uses a control transfer mechanism that is simpler than that used in conventional RPC systems. For example, it uses a special threads scheduling mechanism, called handoff scheduling (details of the threads and handoff scheduling mechanism are given in Chapter 8), for direct context switch from the client thread to the server thread of an LRPC. In this mechanism, when a client calls a server's procedure, it provides the server with an argument stack and its own thread of execution. The call causes a trap to the kernel. The kernel validates the caller, creates a call linkage, and dispatches the client's thread directly to the server domain, causing the server to start executing immediately. When the called procedure completes, control and results return through the kernel back to the point of the client's call. In contrast to this, in conventional RPC implementations, context switching between the client and server threads of an RPC is slow because the client thread and the server thread are fixed in their own domains, signaling one another at a rendezvous, and the

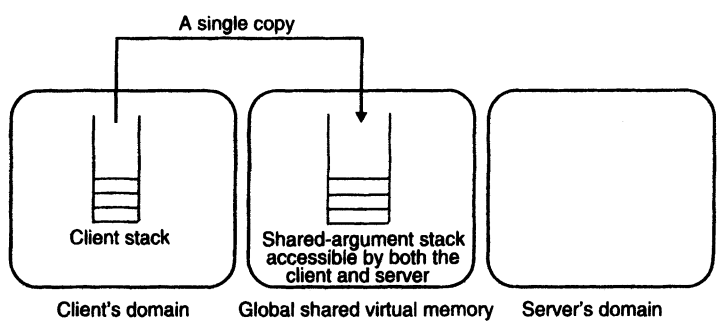
scheduler must manipulate system data structures to block the client's thread and then select one of the server's threads for execution.

4.18.2 Simple Data Transfer

In an RPC, arguments and results need to be passed between the client and server domains in the form of messages. As compared to traditional RPC systems, LRPC reduces the cost of data transfer by performing fewer copies of the data during its transfer from one domain to another. For example, let us consider the path taken by a procedure's argument during a traditional cross-domain RPC. As shown in Figure 4.14(*a*), an argument in this case normally has to be copied four times:



(*a*)



(*b*)

Fig. 4.14 Data transfer mechanisms in traditional cross-domain RPC and LRPC. (*a*) The path taken by a procedure's argument during a traditional cross-domain RPC involves four copy operations. (*b*) The path taken by a procedure's argument during LRPC involves a single-copy operation.

1. From the client's stack to the RPC message
2. From the message in the client domain to the message in the kernel domain
3. From the message in the kernel domain to the message in the server domain
4. From the message in the server domain to the server's stack

To simplify this data transfer operation, LRPC uses a shared-argument stack that is accessible to both the client and the server. Therefore, as shown in Figure 4.14(*b*), the same argument in an LRPC can be copied only once—from the client's stack to the shared-argument stack. The server uses the argument from the argument stack. Pairwise allocation of argument stacks enables LRPC to provide a private channel between the client and server and also allows the copying of parameters and results as many times as are necessary to ensure correct and safe operation.

4.18.3 Simple Stubs

The distinction between cross-domain and cross-machine calls is usually made transparent to the stubs by lower levels of the RPC system. This results in an interface and execution path that are general but infrequently needed.

The use of a simple model of control and data transfer in LRPC facilitates the generation of highly optimized stubs. Every procedure has a call stub in the client's domain and an entry stub in the server's domain. A three-layered communication protocol is defined for each procedure in an LRPC interface:

1. End to end, described by the calling conventions of the programming language and architecture
2. Stub to stub, implemented by the stubs themselves
3. Domain to domain, implemented by the kernel

To reduce the cost of interlayer crossings, LRPC stubs blur the boundaries between the protocol layers. For example, at the time of transfer of control, the kernel associates execution stacks with the initial call frame expected by the called server's procedure and directly invokes the corresponding procedure's entry in the server's domain. No intermediate message examination or dispatching is done, and the server stub starts executing the procedure by directly branching to the procedure's first instruction. Notice that with this arrangement a simple LRPC needs only one formal procedure call (into the client stub) and two returns (one out of the server procedure and one out of the client stub).

4.18.4 Design for Concurrency

When the node of the client and server processes of an LRPC has multiple processors with a shared memory, special mechanisms are used to achieve higher call throughput and lower call latency than is possible on a single-processor node. Throughput is increased by avoiding needless lock contention by minimizing the use of shared-data structures on the

critical domain transfer path. On the other hand, latency is reduced by reducing context-switching overhead by caching domains on idle processors. This is basically a generalization of the idea of decreasing operating system latency by caching recently blocked threads on idle processors to reduce wake-up latency. Instead of threads, LRPC caches domains so that any thread that needs to run in the context of an idle domain can do so quickly, not just the thread that ran there most recently.

Based on the performance evaluation made by Bershad et al. [1990], it was found that LRPC achieves a factor-of-three performance improvement over more traditional approaches. Thus LRPC reduces the cost of cross-domain communication to nearly the lower bound imposed by conventional hardware.

4.19 OPTIMIZATIONS FOR BETTER PERFORMANCE

As with any software design, performance is an issue in the design of a distributed application. The description of LRPC shows some optimizations that may be adopted for better performance of distributed applications using RPC. Some other optimizations that may also have significant payoff when adopted for designing RPC-based distributed applications are described below.

4.19.1 Concurrent Access to Multiple Servers

Although one of the benefits of RPC is its synchronization property, many distributed applications can benefit from concurrent access to multiple servers. One of the following three approaches may be used for providing this facility:

1. The use of threads (described in Chapter 8) in the implementation of a client process where each thread can independently make remote procedure calls to different servers. This method requires that the addressing in the underlying protocol is rich enough to provide correct routing of responses.

2. Another method is the use of the early reply approach [Wilbur and Bacarisse 1987]. As shown in Figure 4.15, in this method a call is split into two separate RPC calls, one passing the parameters to the server and the other requesting the result. In reply to the first call, the server returns a tag that is sent back with the second call to match the call with the correct result. The client decides the time delay between the two calls and carries out other activities during this period, possibly making several other RPC calls. A drawback of this method is that the server must hold the result of a call until the client makes a request for it. Therefore, if the request for results is delayed, it may cause congestion or unnecessary overhead at the server.

3. The third approach, known as the call buffering approach, was proposed by Gimson [1985]. In this method, clients and servers do not interact directly with each other. They interact indirectly via a call buffer server. To make an RPC call, a client sends its call request to the call buffer server, where the request parameters together with the name of the server and the client are buffered. The client can then perform other activities until it

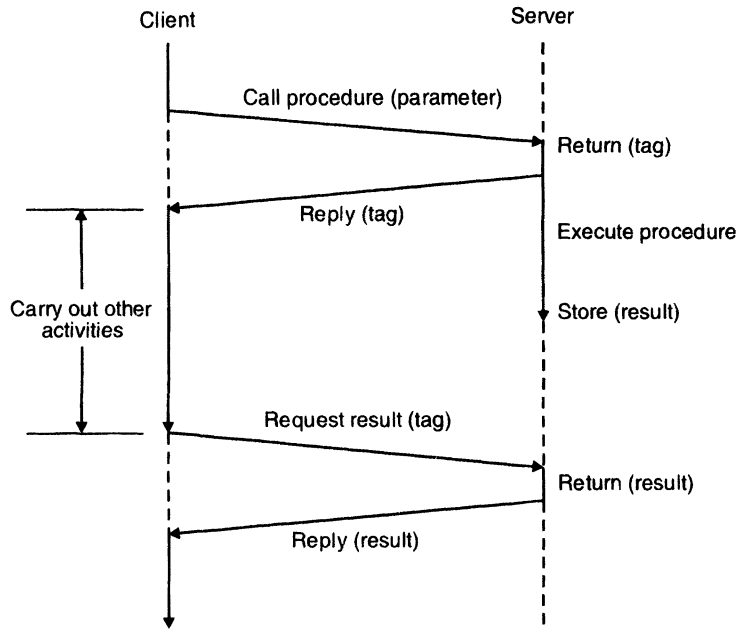


Fig. 4.15 The early reply approach for providing the facility of concurrent access to multiple servers.

needs the result of the RPC call. When the client reaches a state in which it needs the result, it periodically polls the call buffer server to see if the result of the call is available, and if so, it recovers the result. On the server side, when a server is free, it periodically polls the call buffer server to see if there is any call for it. If so, it recovers the call request, executes it, and makes a call back to the call buffer server to return the result of execution to the call buffer server. The method is illustrated in Figure 4.16.

A variant of this approach is used in the Mercury communication system developed at MIT [Liskov and Shrira 1988] for supporting asynchronous RPCs. The Mercury communication system has a new data type called *promise* that is created during an RPC call and is given a type corresponding to those of the results and exceptions of the remote procedure. When the results arrive, they are stored in the appropriate promise, from where the caller claims the results at a time suitable to it. Therefore, after making a call, a caller can continue with other work and subsequently pick up the results of the call from the appropriate promise.

A promise is in one of two states—blocked or ready. It is in a blocked state from the time of creation to the time the results of the call arrive, whereupon it enters the ready state. A promise in the ready state is immutable.

Two operations (*ready* and *claim*) are provided to allow a caller to check the status of the promise for the call and to claim the results of the call from it. The *ready* operation is used to test the status (blocked/ready) of the promise. It returns true or false according to whether the promise is ready or blocked. The *claim* operation is

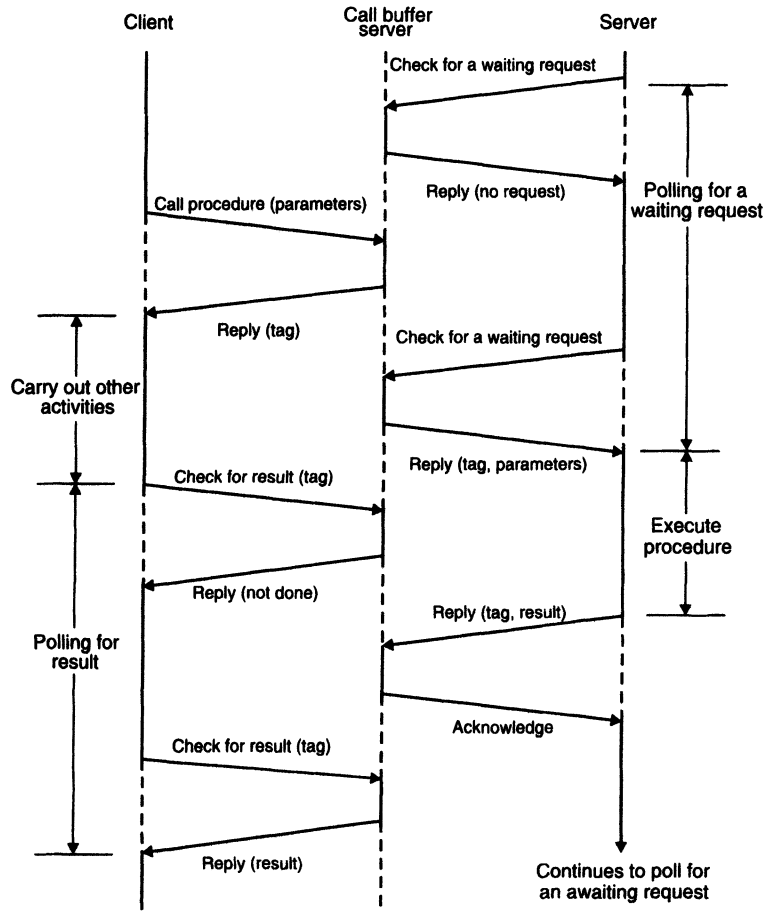


Fig. 4.16 The call buffering approach for providing the facility of concurrent access to multiple servers.

used to obtain the results of the call from the promise. The *claim* operation blocks the caller until the promise is ready, whereupon it returns the results of the call. Therefore, if the caller wants to continue with other work until the promise becomes ready, it can periodically check the status of the promise by using the *ready* operation.

4.19.2 Serving Multiple Requests Simultaneously

The following types of delays are commonly encountered in RPC systems:

1. Delay caused while a server waits for a resource that is temporarily unavailable. For example, during the course of a call execution, a server might have to wait for accessing a shared file that is currently locked elsewhere.

2. A delay can occur when a server calls a remote function that involves a considerable amount of computation to complete or involves a considerable transmission delay.

For better performance, good RPC implementations must have mechanisms to allow the server to accept and process other requests, instead of being idle while waiting for the completion of some operation. This requires that a server be designed in such a manner that it can service multiple requests simultaneously. One method to achieve this is to use the approach of a multiple-threaded server with dynamic threads creation facility for server implementation (details of this approach are given in Chapter 8).

4.19.3 Reducing Per-Call Workload of Servers

Numerous client requests can quickly affect a server's performance when the server has to do a lot of processing for each request. Thus, to improve the overall performance of an RPC system, it is important to keep the requests short and the amount of work required by a server for each request low. One way of accomplishing this improvement is to use stateless servers and let the clients keep track of the progression of their requests sent to the servers. This approach sounds reasonable because, in most cases, the client portion of an application is really in charge of the flow of information between a client and a server.

4.19.4 Reply Caching of Idempotent Remote Procedures

The use of a reply cache to achieve exactly-once semantics in nonidempotent remote procedures has already been described. However, a reply cache can also be associated with idempotent remote procedures for improving a server's performance when it is heavily loaded. When client requests to a server arrive at a rate faster than the server can process the requests, a backlog develops, and eventually client requests start timing out and the clients resend the requests, making the problem worse. In such a situation, the reply cache helps because the server has to process a request only once. If a client resends its request, the server just sends the cached reply.

4.19.5 Proper Selection of Timeout Values

To deal with failure problems, timeout-based retransmissions are necessary in distributed applications. An important issue here is how to choose the timeout value. A "too small" timeout value will cause timers to expire too often, resulting in unnecessary retransmissions. On the other hand, a "too large" timeout value will cause a needlessly long delay in the event that a message is actually lost. In RPC systems, servers are likely to take varying amounts of time to service individual requests, depending on factors such as server load, network routing, and network congestion. If clients continue to retry sending those requests for which replies have not yet been received, the server loading and network congestion problem will become worse. To prevent this situation, proper selection of timeout values is important. One method to handle this issue is to use some sort of back-off strategy of exponentially increasing timeout values.

4.19.6 Proper Design of RPC Protocol Specification

For better performance, the protocol specification of an RPC system must be properly designed so as to minimize the amount of data that has to be sent over the network and the frequency at which it is sent. Reducing the amount of data to be transferred helps in two ways: It requires less time to encode and decode the data and it requires less time to transmit the data over the network. Several existing RPC systems use TCP/IP or UDP/IP as the basic protocol because they are easy to use and fit in well with existing UNIX systems and networks such as the Internet. This makes it straightforward to write clients and servers that run on UNIX systems and standard networks. However, the use of a standard general-purpose protocol for RPC generally leads to poor performance because general-purpose protocols have many features to deal with different problems in different situations. For example, packets in the IP suite (to which TCP/IP and UDP/IP belong) have in total 13 header fields, of which only 3 are useful for an RPC—the source and destination addresses and the packet length. However, several of these header fields, such as those dealing with fragmentation and checksum, have to be filled in by the sender and verified by the receiver to make them legal IP packets. Some of these fields, such as the checksum field, are time consuming to compute. Therefore, for better performance, an RPC system should use a specialized RPC protocol. Of course, a new protocol for this purpose has to be designed from scratch, implemented, tested, and embedded in existing systems, so it requires considerably more work.

4.20 CASE STUDIES: SUN RPC, DCE RPC

Many RPC systems have been built and are in use today. Notable ones include the Cedar RPC system [Birrell and Nelson 1984], Courier in the Xerox NS family of protocols [Xerox Corporation 1981], the Eden system [Almes et al. 1985], the CMU Spice system [Jones et al. 1985], Sun RPC [Sun Microsystems 1985], Argus [Liskov and Scheifler 1983], Arjuna [Shrivastava et al. 1991], the research system built at HP Laboratories [Gibbons 1987], NobelNet's EZ RPC [Smith 1994], Open Software Foundation's (OSF's) DCE RPC [Rosenberry et al. 1992], which is a descendent of Apollo's Network Computing Architecture (NCA), and the HRPC system developed at the University of Washington [Bershad et al. 1987]. Of these, the best known UNIX RPC system is the Sun RPC. Therefore, the Sun RPC will be described in this section as a case study. In addition, due to the policy used in this book to describe DCE components as case studies, the DCE RPC will also be briefly described.

4.20.1 Sun RPC

Stub Generation

Sun RPC uses the automatic stub generation approach, although users have the flexibility of writing the stubs manually. An application's interface definition is written in an IDL called RPC Language (RPCL). RPCL is an extension of the Sun XDR

language that was originally designed for specifying external data representations. As an example, the interface definition of the stateless file service, described in Section 4.8.1, is given in Figure 4.17. As shown in the figure, an interface definition contains a program number (which is 0 x 20000000 in our example) and a version number of the service (which is 1 in our example), the procedures supported by the service (in our example READ and WRITE), the input and output parameters along with their types for each procedure, and the supporting type definitions. The three values program number (STATELESS_FS_PROG), version number (STATELESS_FS_VERS), and a procedure number (READ or WRITE) uniquely identify a remote procedure. The READ and WRITE procedures are given numbers 1 and 2, respectively. The number 0 is reserved for a null procedure that is automatically generated and is intended to be used to test whether a server is available. Interface definition file names have an extension *.x*. (for example, *StatelessFS.x*).

```
/* Interface definition for a stateless file service (StatelessFS)
   in file StatelessFS.x */
const FILE_NAME_SIZE = 16
const BUFFER_SIZE = 1024

typedef string FileName<FILE_NAME_SIZE>;
typedef long Position;
typedef long Nbytes;

struct Data {
        long n;
        char buffer[BUFFER_SIZE];
};

struct readargs {
        FileName        filename;
        Position        position;
        Nbytes          n;
};

struct writeargs {
        FileName        filename;
        Position        position;
        Data            data;
};

program STATELESS_FS_PROG {
        version STATELESS_FS_VERS {
                        Data        READ (readargs) = 1;
                        Nbytes      WRITE (writeargs) = 2;
        } = 1;
} = 0x20000000;
```

Fig. 4.17 Interface definition for a stateless file service written in RPCL of Sun RPC.

The IDL compiler is called *rpcgen* in Sun RPC. From an interface definition file, *rpcgen* generates the following:

1. A header file that contains definitions of common constants and types defined in the interface definition file. It also contains external declarations for all XDR marshaling and unmarshaling procedures that are automatically generated. The name of the header file is formed by taking the base name of the input file to *rpcgen* and adding a *.h* suffix (for example, *StatelessFS.h*). This file is manually included in client and server program files and automatically included in client stub, server stub, and XDR filters files using *#include*.

2. An XDR filters file that contains XDR marshaling and unmarshaling procedures. These procedures are used by the client and server stub procedures. The name of this file is formed by taking the base name of the input file to *rpcgen* and adding a *_xdr.c* suffix (for example, *StatelessFS_xdr.c*).

3. A client stub file that contains one stub procedure for each procedure defined in the interface definition file. A client stub procedure name is the name of the procedure given in the interface definition, converted to lowercase and with an underscore and the version number appended. For instance, in our example, the client stub procedure names for *READ* and *WRITE* procedures will be *read_1* and *write_1*, respectively. The name of the client stub file is formed by taking the base name of the input file to *rpcgen* and adding a *_clnt.c* suffix (for example, *StatelessFS_clnt.c*).

4. A server stub file that contains the *main* routine, the *dispatch* routine, and one stub procedure for each procedure defined in the interface definition file plus a null procedure.

The *main* routine creates the transport handles and registers the service. The default is to register the program on both the UDP and TCP transports. However, a user can select which transport to use with a command-line option to *rpcgen*.

The *dispatch* routine dispatches incoming remote procedure calls to the appropriate procedure. The name used for the dispatch routine is formed by mapping the program name to lowercase characters and appending an underscore followed by the version number (for example, *stateless_fs_prog_1*).

The name of the server stub file is formed by taking the base name of the input file to *rpcgen* and adding a *_svc.c* suffix (for example, *StatelessFS_svc.c*).

Now using the files generated by *rpcgen*, an RPC application is created in the following manner:

1. The application programmer manually writes the client program and server program for the application. The skeletons of these two programs for our example application of stateless file service are given Figures 4.18 and 4.19, respectively. Notice that the remote procedure names used in these two programs are those of the stub procedures (*read_1* and *write_1*).

```
/* A skeleton of client source program for the stateless file service in file client.c */
#include <stdio.h>
#include <rpc/rpc.h>
#include "StatelessFS.h"

main (argc, argv)
        int argc;
        char **argv;
{
        CLIENT          *client_handle;
        char            *server_host_name = "paris";
        readargs        read_args;
        writeargs       write_args;
        Data            *read_result;
        Nbytes          *write_result;

        client_handle = clnt_create (server_host_name, STATELESS_FS_PROG,
                        STATELESS_FS_VERS, "udp");
                        /* Get a client handle. Creates socket */
        if (client_handle == NULL) {
                        clnt_pcreateerror (server_host_name);
                        return (1); /* Cannot contact server */
        };

        /* Prepare parameters and make an RPC to read procedure */
        read_args.filename = "example";
        read_args.position = 0;
        read_args.n = 500;
        read_result = read_1 (&read_args, client_handle);
        . . .
        . . .
        . . .

        /* Prepare parameters and make an RPC to write procedure */
        write_args.filename = "example";
        write_args.position = 501;
        write_args.data.n = 100;
        /* Statements for putting 100 bytes of data in &write_args.data.buffer */
        write_result = write_1 (&write_args, client_handle);
        . . .
        . . .
        . . .
        clnt_destroy (client_handle);
        /* Destroy the client handle when done. Closes socket */
}
```

Fig. 4.18 A skeleton of client source program for the stateless file service of Figure 4.17.

```
/* A skeleton of server source program for the stateless file service in file server.c */
#include <stdio.h>
#include <rpc/rpc.h>
#include "StatelessFS.h"

/* READ PROCEDURE */
Data    *read_1 (args) /* Input parameters as a single argument */
        readargs        *args;
{
        static Data result; /* Must be declared as static */

        /* Statements for reading args.n bytes from the file args.filename starting
        from position args.position, and for putting the data read in &result.buffer
        and the actual number of bytes read in result.n */

        return (&result); /* Return the result as a single argument */
}

/* WRITE PROCEDURE */
Nbytes  *write_1 (args) /* Input parameters as a single argument */
        writeargs       *args;
{
        static Nbytes result; /* Must be declared as static */

        /* Statements for writing args.data.n bytes of data from the buffer
        &args.data.buffer into the file args.filename starting at position
        args.position */

        /* Statement for putting the actual number of bytes written in result */

        return (&result);
}
```

Fig. 4.19 A skeleton of server source program for the stateless file service of Figure 4.17.

2. The client program file is compiled to get a client object file.
3. The server program file is compiled to get a server object file.
4. The client stub file and the XDR filters file are compiled to get a client stub object file.
5. The server stub file and the XDR filters file are compiled to get a server stub object file.
6. The client object file, the client stub object file, and the client-side RPCRuntime library are linked together to get the client executable file.
7. The server object file, the server stub object file, and the server-side RPCRuntime library are linked together to get the server executable file.

The entire process is summarized in Figure 4.20.

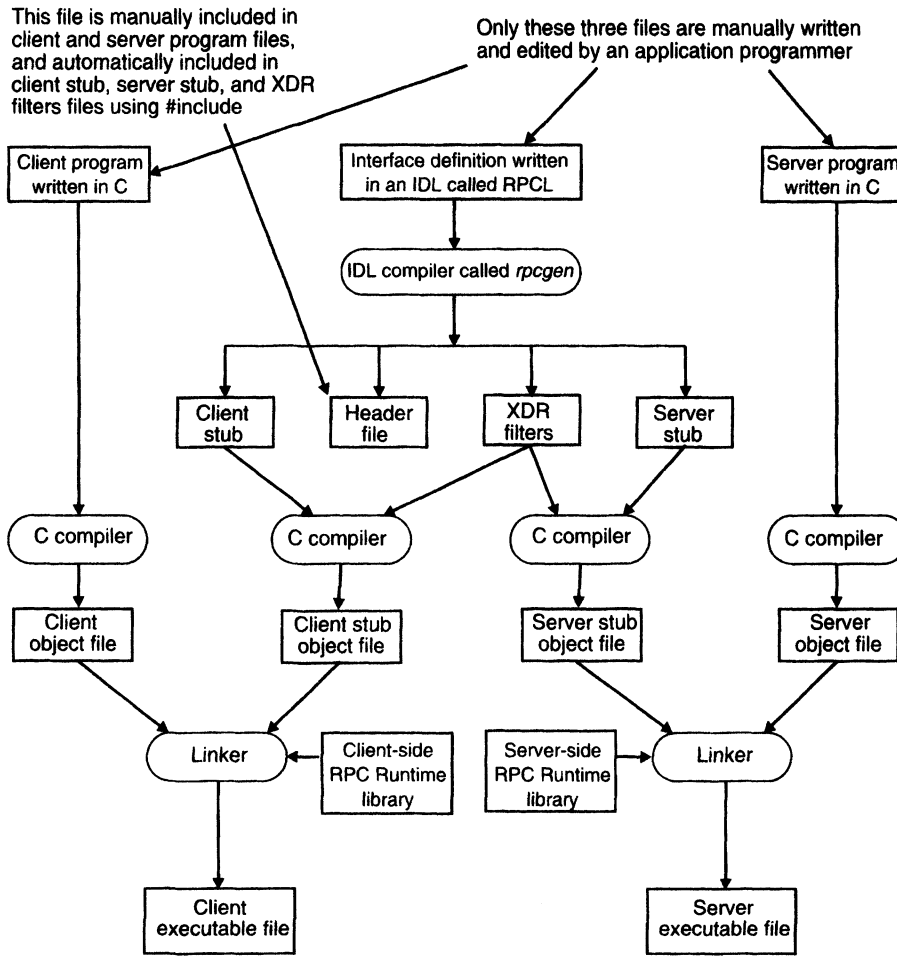


Fig. 4.20 The steps in creating an RPC application in Sun RPC.

Procedure Arguments

In Sun RPC, a remote procedure can accept only one argument and return only one result. Therefore, procedures requiring multiple parameters as input or as output must include them as components of a single structure. This is the reason why the structures *Data* (used as a single output argument to the *READ* procedure), *readargs* (used as a single input argument to the *READ* procedure), and *writeargs* (used as a single input argument to the *WRITE* procedure) have been defined in our example of Figures 4.17–4.19. If a remote procedure does not take an argument, a NULL pointer must still be passed as an argument to the remote procedure. Therefore, a Sun RPC call always has two arguments—the first is a pointer to the single argument of the remote procedure and the second is a pointer to a client handle (see the calls for *read_1* and *write_1* in Fig. 4.18). On the other hand, a

return argument of a procedure is a pointer to the single result. The returned result must be declared as a static variable in the server program because otherwise the value of the returned result becomes undefined when the procedure returns (see the return argument *result* in Fig. 4.19).

Marshaling Arguments and Results

We have seen that Sun RPC allows arbitrary data structures to be passed as arguments and results. Since significant data representation differences can exist between the client computer and the server computer, these data structures are converted to eXternal Data Representation (XDR) and back using marshaling procedures. The marshaling procedures to be used are specified by the user and may be either built-in procedures supplied in the RPCRuntime library or user-defined procedures defined in terms of the built-in procedures. The RPCRuntime library has procedures for marshaling integers of all sizes, characters, strings, reals, and enumerated types.

Since XDR encoding and decoding always occur, even between a client and server of the same architecture, unnecessary overhead is added to the network service for those applications in which XDR encoding and decoding are not needed. In such cases, user-defined marshaling procedures can be utilized. That is, users can write their own marshaling procedures verifying that the architectures of the client and the server machines are the same and, if so, use the data without conversion. If they are not the same, the correct XDR procedures can be invoked.

Call Semantics

Sun RPC supports at-least-once semantics. After sending a request message, the RPCRuntime library waits for a timeout period for the server to reply before retransmitting the request. The number of retries is the total time to wait divided by the timeout period. The total time to wait and the timeout period have default values of 25 and 5 seconds, respectively. These default values can be set to different values by the users. Eventually, if no reply is received from the server within the total time to wait, the RPCRuntime library returns a timeout error.

Client-Server Binding

Sun RPC does not have a networkwide binding service for client-server binding. Instead, each node has a local binding agent called *portmapper* that maintains a database of mapping of all local services (as already mentioned, each service is identified by its program number and version number) and their port numbers. The portmapper runs at a well-known port number on every node.

When a server starts up, it registers its program number, version number, and port number with the local portmapper. When a client wants to do an RPC, it must first find out the port number of the server that supports the remote procedure. For this, the client makes a remote request to the portmapper at the server's host, specifying the program number and version number (see *clnt_create* part of Fig. 4.18). This means that a client

must specify the host name of the server when it imports a service interface. In effect, this means that Sun RPC has no location transparency.

The procedure *clnt_create* is used by a client to import a service interface. It returns a client handle that contains the necessary information for communicating with the corresponding server port, such as the socket descriptor and socket address. The client handle is used by the client to directly communicate with the server when making subsequent RPCs to procedures of the service interface (see RPCs made to *read_1* and *write_1* procedures in Fig. 4.18).

Exception Handling

The RPCRuntime library of Sun RPC has several procedures for processing detected errors. The server-side error-handling procedures typically send a reply message back to the client side, indicating the detected error. However, the client-side error-handling procedures provide the flexibility to choose the error-reporting mechanism. That is, errors may be reported to users either by printing error messages to *stderr* or by returning strings containing error messages to clients.

Security

Sun RPC supports the following three types of authentication (often referred to as *flavors*):

1. *No authentication.* This is the default type. In this case, no attempt is made by the server to check a client's authenticity before executing the requested procedure. Consequently, clients do not pass any authentication parameters in request messages.

2. *UNIX-style authentication.* This style is used to restrict access to a service to a certain set of users. In this case, the *uid* and *gid* of the user running the client program are passed in every request message, and based on this authentication information, the server decides whether to execute the requested procedure or not.

3. *DES-style authentication.* Data Encryption Standard (DES) is an encryption technique described in Chapter 11. In DES-style authentication, each user has a globally unique name called *netname*. The *netname* of the user running the client program is passed in encrypted form in every request message. On the server side, the encrypted *netname* is first decrypted and then the server uses the information in *netname* to decide whether to execute the requested procedure or not.

The DES-style authentication is recommended for users who need more security than UNIX-style authentication. RPCs using DES-style authentication are also referred to as *secure RPC*.

Clients have the flexibility to select any of the above three authentication flavors for an RPC. The type of authentication can be specified when a client handle is created. It is possible to use a different authentication mechanism for different remote procedures within a distributed application by setting the authentication type to the flavor desired before doing the RPC.

The authentication mechanism of Sun RPC is open ended in the sense that in addition to the three authentication types mentioned above users are free to invent and use new authentication types.

Special Types of RPCs

Sun RPC provides support for asynchronous RPC, callback RPC, broadcast RPC, and batch-mode RPC.

Asynchronous RPC is accomplished by setting the timeout value of an RPC to zero and writing the server such that no reply is generated for the request.

To facilitate callback RPC, the client registers the callback service using a transient program number with the local portmapper. The program number is then sent as part of the RPC request to the server. The server initiates a normal RPC request to the client using the given program number when it is ready to do the callback RPC.

A broadcast RPC is directed to the portmapper of all nodes. Each node's portmapper then passes it on to the local service with the given program name. The client picks up any replies one by one.

Batch-mode RPC is accomplished by batching of client calls that require no reply and then sending them in a pipeline to the server over TCP/IP.

Critiques of Sun RPC

In spite of its popularity, some of the criticisms normally made against Sun RPC are as follows:

1. Sun RPC lacks location transparency because a client has to specify the host name of the server when it imports a service interface.
2. The interface definition language of Sun RPC does not allow a general specification of procedure arguments and results. It allows only a single argument and a single result. This requirement forces multiple arguments or return values to be packaged as a single structure.
3. Sun RPC is not transport independent and the transport protocol is limited to either UDP or TCP. However, a transport-independent version of Sun RPC, known as TI-RPC (transport-independent RPC), has been developed by Sun-Soft, Inc. TI-RPC provides a simple and consistent way in which transports can be dynamically selected depending upon user preference and the availability of the transport. Details of TI-RPC can be found in [Khanna 1994].
4. In UDP, Sun RPC messages are limited to 8 kilobytes in length.
5. Sun RPC supports only at-least-once call semantics, which may not be acceptable for some applications.
6. Sun RPC does not have a networkwide client-server binding service.
7. We saw in Section 4.18 that threads can be used in the implementation of a client or a server process for better performance of an RPC-based application. Sun RPC does not include any integrated facility for threads in the client or server, although Sun OS has a separate threads package.

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

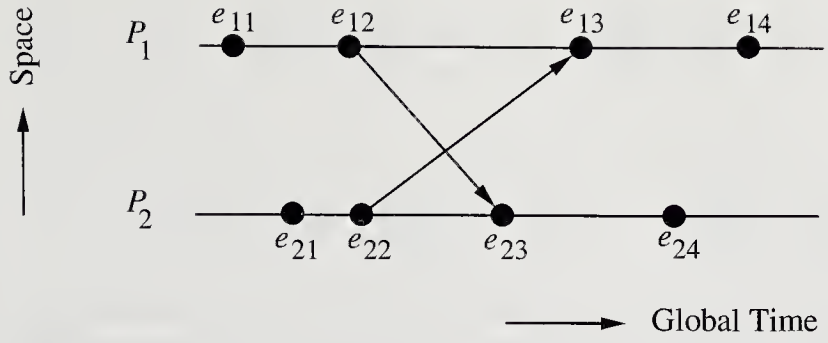


FIGURE 5.2
A space-time diagram.

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

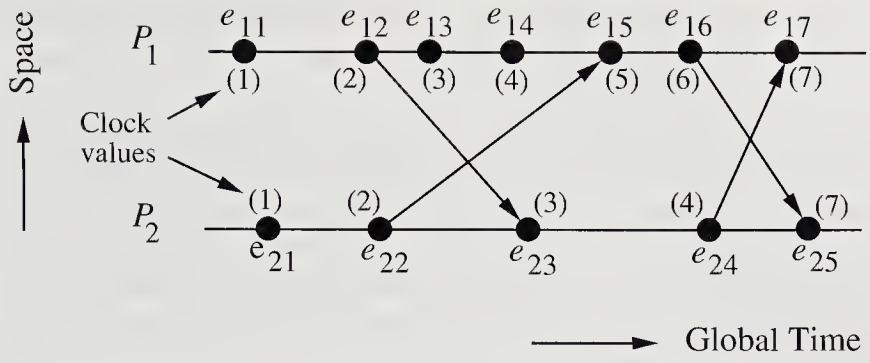


FIGURE 5.3
How Lamport's logical clocks advance.

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.

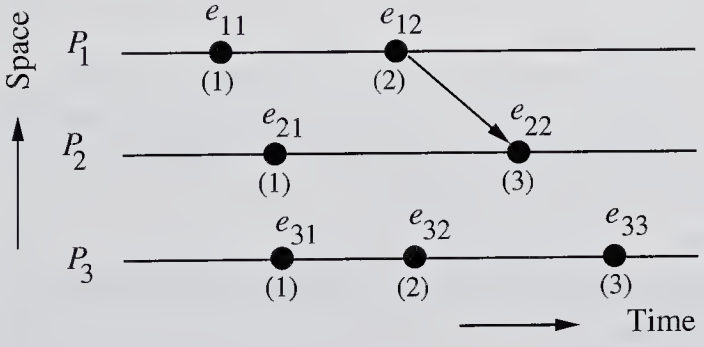


FIGURE 5.4
A space-time diagram.

The reason for the above limitation is that each clock can independently advance due to the occurrence of local events in a process and the Lamport's clock system cannot distinguish between the advancements of clocks due to local events from those due to the exchange of messages between processes. (Notice that only message exchanges establish paths in a space-time diagram between events occurring in different processes.) Therefore, using the timestamps assigned by Lamport's clocks, we cannot reason about the causal relationship between two events occurring in different processes by just looking at the timestamps of the events. In the next section, we present a scheme of vector clocks that gives us the ability to decide whether two events are causally related or not by simply looking at their timestamps.

5.4 VECTOR CLOCKS

The system of vector clocks was independently proposed by Fidge [5] and Mattern [16]. A concept similar to vector clocks was proposed previously by Strom and Yemini [38] for keeping track of transitive dependencies among processes for recovery purposes. Let n be the number of processes in a distributed system. Each process P_i is equipped with a clock C_i, which is an integer vector of length n. The clock C_i can be thought of as a function that assigns a vector $C_i(a)$ to any event a. $C_i(a)$ is referred to as the timestamp of event a at P_i. $C_i[i]$, the ith entry of C_i, corresponds to P_i's own logical time. $C_i[j]$, $j \neq i$ is P_i's best guess of the logical time at P_j. More specifically, at any point in time, the jth entry of C_i indicates the time of occurrence of the last event at P_j which "happened before" the current point in time at P_i. This "happened before" relationship could be established directly by communication from P_j to P_i or indirectly through communication with other processes.

The implementation rules for the vector clocks are as follows [16]:

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

$$C_i[i] := C_i[i] + d \qquad (d > 0) \tag{5.3}$$

[IR2] If event a is the sending of the message m by process P_i, then message m is assigned a vector timestamp $t_m = C_i(a)$; on receiving the same message m by process P_j, C_j is updated as follows:

$$\forall k,\ C_j[k] := \max(C_j[k], t_m[k]) \tag{5.4}$$

Note that, on the receipt of messages, a process learns about the more recent clock values of the rest of the processes in the system.

In rule IR1, we treat message send and message receive by a process as events. In rule IR2, a message is assigned a timestamp after the sender process has incremented its clock due to IR1. If it is necessary to allow for propagation time for a message, then IR2 can be performed after performing the following step [5].

$$\text{If} \quad C_j[i] \leq t_m[i] \quad \text{then} \quad C_j[i] := t_m[i] + d \qquad (d > 0)$$

However, the above step is not necessary to relate events causally and hence, we do not make use of it in the following discussion.

Assertion. At any instant,

$$\forall i,\ \forall j:\ C_i[i] \geq C_j[i]$$

The proof is obvious because no process $P_j \neq P_i$ can have more up-to-date knowledge about the clock value of process i and clocks are monotonically nondecreasing.

Example 5.5. Figure 5.5 illustrates an example of how clocks advance and the dissemination of time occurs in a system using vector elocks (d is assumed to be 1 and all clock values are initially zero).

Event e_{11} is an internal event in process P_1 that causes $C_1[1]$ to be ineremented to 1 due to IR1. e_{12} is a message send event in P_1 which causes $C_1[1]$ to be incremented to 2 due to IR1. e_{22} is a message receive event in P_2 that causes $C_2[2]$ to be incremented to 2 due to IR1, and $C_2[1]$ to be set to 2 due to IR2. e_{31} is a send event in P_3 which causes $C_3[3]$ to be incremented to 1 due to IR1. Event e_{23}, a receive event in P_2, causes $C_2[2]$ to be incremented to 3 due to IR1, and $C_2[3]$ to be set to 1 due to IR2. e_{24} is a send event in P_2 and e_{13} is the corresponding receive event. Note that $C_1[3]$ is set to 1 due to IR2, and process P_1 has learned that the local clock value at P_3 is at least 1 through a message from P_2.

Vector timestamps can be compared as follows [16]. For any two vector timestamps t^a and t^b of events a and b, respectively:

Equal:	$t^a = t^b$	iff	$\forall i,\ t^a[i] = t^b[i]$;
Not Equal:	$t^a \neq t^b$	iff	$\exists i,\ t^a[i] \neq t^b[i]$;
Less Than or Equal:	$t^a \leq t^b$	iff	$\forall i,\ t^a[i] \leq t^b[i]$;
Not Less Than or Equal To:	$t^a \not\leq t^b$	iff	$\exists i,\ t^a[i] > t^b[i]$;
Less Than:	$t^a < t^b$	iff	$(t^a \leq t^b \ \wedge \ t^a \neq t^b)$;
Not Less Than:	$t^a \not< t^b$	iff	$\neg(t^a \leq t^b \ \wedge \ t^a \neq t^b)$;
Concurrent:	$t^a \| t^b$	iff	$(t^a \not< t^b \ \wedge \ t^b \not< t^a)$;

Note that the relation "$\leq$" is a partial order. However, the relation "$\|$" is not a partial order because it is not transitive.

CAUSALLY RELATED EVENTS. Events a and b are causally related, if $t^a < t^b$ or $t^b < t^a$. Otherwise, these events are concurrent.

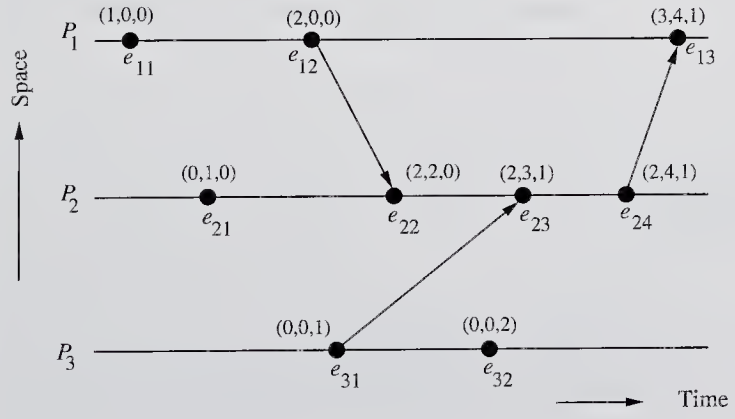


FIGURE 5.5
Dissemination of time in virtual clocks.

THE PROTOCOL

Sending of a message M from process P_1 to process P_2

- Send message M (timestamped t_M) along with V_P_1 to process P_2.
- Insert pair (P_2, t_M) into V_P_1. If V_P_1 contains a pair (P_2, t), it simply gets overwritten by the new pair (P_2, t_M). Note that the pair (P_2, t_M) was not sent to P_2. Any future message carrying the pair (P_2, t_M) cannot be delivered to P_2 until $t_M < t_{P_2}$.

Arrival of a message M at process P_2

If V_M (the vector accompanying message M) does not contain any pair (P_2, t)
then the message can be delivered
else (* A pair (P_2, t) exists in V_M *)
If $t \not< t_{P_2}$ then
the message cannot be delivered (*it is buffered for later delivery*)
else
the message can be delivered.

If message M can be delivered at process P_2, then the following three actions are taken:

1. Merge V_M accompanying M with V_P_2 in the following manner:
   - If $(\exists\ (P, t) \in V\_M$, sueh that $P \neq P_2)$ and $(\forall (P', t) \in V\_P_2.\ P' \neq P)$, then insert (P, t) into V_P_2. This rule performs the following: if there is no entry for process P in V_P_2, and V_M contains an entry for process P, insert that entry into V_P_2.
   - $\forall P, P \neq P_2$, if $((P, t) \in V\_M) \wedge ((P, t') \in V\_P_2)$, then the algorithm takes the following actions: $(P, t) \in V\_P_2$ can be substituted by the pair (P, t_{sup}) where t_{sup} is such that $\forall i, t_{\text{sup}}[i] = \max(t[i], t'[i])$. This rule is simply performing the step in Eq. 5.4 for each entry in V_P2.

     Due to the above two actions, the algorithm satisfies the following two conditions:

   a. No message can be delivered to P as long as $t' < t_P$ is not true.
   b. No message can be delivered to P as long as $t < t_P$ is not true.
2. Update site P_2's logical clock.
3. Check for the buffered messages that can now be delivered since local clock has been updated.

A pair (P, t) can be deleted from the vector maintained at a site after ensuring that the pair (P, t) has become obsolete (i.e., no longer needed) (see Problem 5.3).

5.6 GLOBAL STATE

We now address the problem of collecting or recording a coherent (consistent) global state in distributed systems, a challenging task due to the absence of a global clock and

shared memory. First, we reexamine the bank account example of Sec. 5.2 to develop the correctness criteria for a consistent global state recording algorithm.

Figure 5.7 shows the stages of a computation when \$50 is transferred from account A to account B. The communication channels C1 and C2 are assumed to be FIFO.

Suppose the state of account A at site S1 was recorded when the global state was 1 (see Fig. 5.7). Now assume that the global state changes to 2, and the state of communication channels C1 and C2 and of account B are recorded when the global state is 2. Then the composite of all the states recorded would show account A's balance as \$500, account B's balance as \$200, and a message in transit to transfer \$50. In other words, an extra amount of \$50 would appear in the global state. The reason for this inconsistency is that A's state was recorded before the message was sent and the channel C1's state was recorded after the message was sent. Therefore, a recorded global state may be inconsistent if $n < n'$ where n is the number of messages sent by A along

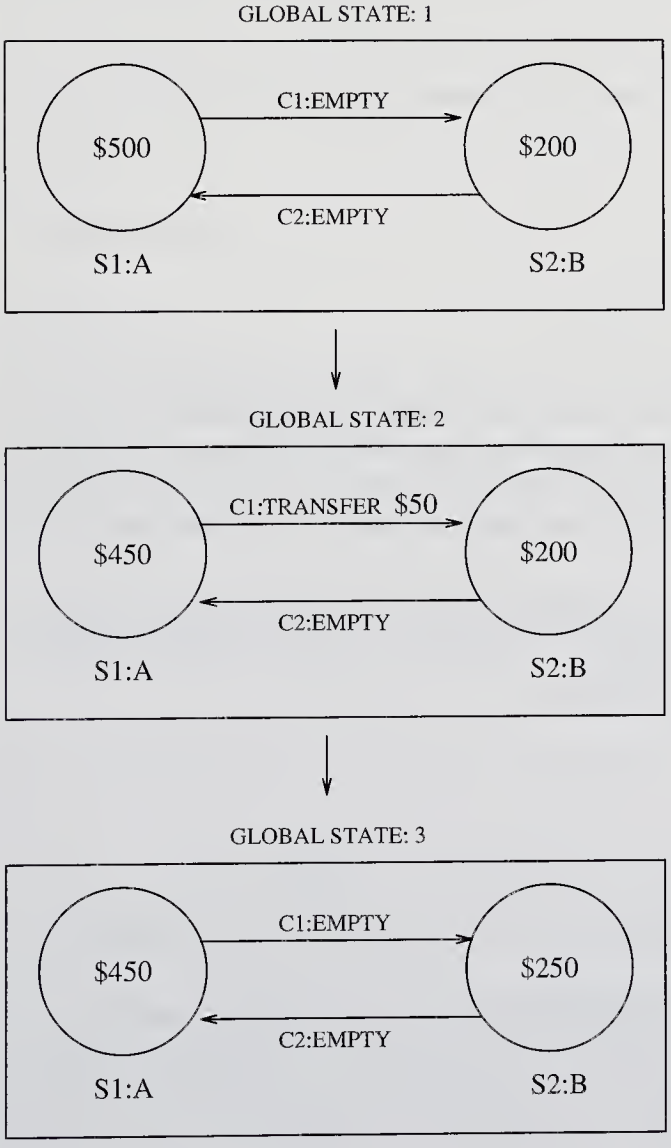


FIGURE 5.7
Global-states and their transitions in the bank accounts example.

the channel before A's state was recorded and n' is the number of messages sent by A along the channel before the channel's state was recorded.

Suppose channels state were recorded when the global state was 1 and A and B's states were recorded when the global state was 2. Then the composite of A, B, and the channels state will show a deficit of $50. This means that the recorded global state may be inconsistent if $n > n'$. Hence, a consistent global state requires

$$n = n' \tag{5.6}$$

On similar lines, one can show that a consistent global state requires

$$m = m' \tag{5.7}$$

where m' = number of messages received along the channel before account B's state was recorded and m = number of messages received along the channel by B before the channel's state was recorded.

Since in no system the number of messages sent along the channel can be less than the number of messages received along that channel, we have

$$n' \geq m \tag{5.8}$$

From Eqs. 5.6 and 5.8, we get

$$n \geq m \tag{5.9}$$

Therefore, a consistent global state must satisfy Eq. 5.9 . In other words, the state of a communication channel in a consistent global state should be the sequence of messages sent along that channel before the sender's state was recorded, excluding the sequence of messages received along that channel before the receiver's state was recorded [8].

The above observations result in a simple algorithm (described in Sec. 5.6.1) for recording a consistent global state. Before describing the algorithm, some definitions for formally describing a system state are given.

Definitions

LOCAL STATE. For a site (computer) S_i, its local state at a given time is defined by the local eontext of the distributed application [7]. Let LS_i denote the local state of S_i at any time.

Let $send(m_{ij})$ denote the send event of a message m_{ij} by S_i to S_j, and $rec(m_{ij})$ denote the receive event of message m_{ij} by site S_j. Let $time(x)$ denote the time at which state x was recorded and $time(send(m))$ denote the time at which event $send(m)$ occurred.

For a message m_{ij} sent by S_i to S_j, we say that

- $send(m_{ij}) \in LS_i$ iff $time(send(m_{ij})) < time(LS_i)$.
- $rec(m_{ij}) \in LS_j$ iff $time(rec(m_{ij})) < time(LS_j)$.

For the local states LS_i and LS_j of any two sites S_i and S_j, we define two sets of messages. These sets contain messages that were sent from site S_i to site S_j.

Transit: $transit(LS_i,\ LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$
Inconsistent: $inconsistent(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \notin LS_i \wedge rec(m_{ij}) \in LS_j\}$

GLOBAL STATE. A global state, GS, of a system is a collection of the local states of its sites; That is, $GS = \{LS_1,\ LS_2, \ldots, LS_n\}$ where n is the number of sites in the system.

Note that any collection of local states of sites need not represent a consistent global state. Consistency has a connotation that for every effect or outcome recorded in a global state, the cause of the effect must also be recorded in the global state. We next give definitions characterizing global states.

Consistent global state. A global state $GS = \{LS_1,\ LS_2, \ldots, LS_n\}$ is *consistent* iff

$$\forall i, \forall j : 1 \leq i, j \leq n :: inconsistent(LS_i, LS_j) = \Phi$$

Thus, in a consistent global state, for every received message a corresponding send event is recorded in the global state. In an inconsistent global state, there is at least one message whose receive event is recorded but its send event is not recorded in the global state. In Fig. 5.8, the global state $\{LS_{12}, LS_{23}, LS_{33}\}$ and $\{LS_{11}, LS_{22}, LS_{32}\}$ correspond to consistent and inconsistent global states, respectively.

Transitless global state. A global state is *transitless* if and only if

$$\forall i, \forall j : 1 \leq i, j \leq n :: transit(LS_i, LS_j) = \Phi$$

Thus, all communication channels are empty in a transitless global state.

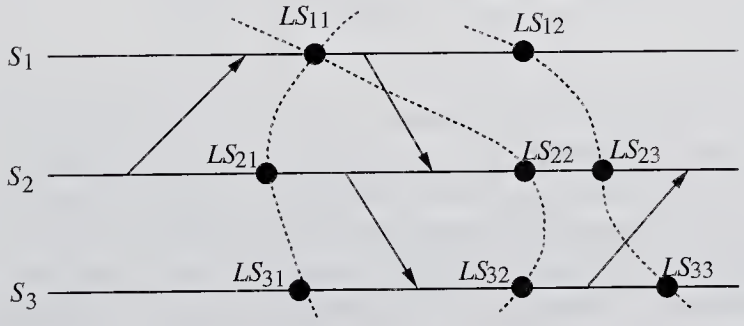


FIGURE 5.8
Global states in a distributed computation.

Strongly consistent global state. A global state is *strongly consistent* if it is consistent and transitless. In a strongly consistent state, not only the send events of all the recorded received events are recorded, but the receive events of all the recorded send events are also recorded. Thus, a strongly consistent state corresponds to a consistent global state in which all channels are empty. In Fig. 5.8, the global state $\{LS_{11}, LS_{21}, LS_{31}\}$ is a strongly consistent global state.

A note. While the definitions of this section are defined for a group of sites, they can also be applied to a group of cooperating processes by simply replacing sites with processes in the definitions. For instance, $GS = \{LS_1, LS_2, \ldots, LS_n\}$ represents a global state of n cooperating processes, where LS_i is the local state of process P_i.

5.6.1 Chandy-Lamport's Global State Recording Algorithm

Chandy and Lamport [8] were the first to propose a distributed algorithm to capture a consistent global state. The algorithm uses a marker (a special message) to initiate the algorithm and the marker has no effect on the underlying computation. The communication channels are assumed to be FIFO. The recorded global state is also referred to as a *snapshot* of the system state.

Marker Sending Rule for a process P

- P records its state.
- For each outgoing channel C from P on which a marker has not been already sent, P sends a marker along C before P sends further messages along C.

Marker Receiving Rule for a process Q. On receipt of a marker along a channel C:

```
If Q has not recorded its state
    then
        begin
            Record the state of C as an empty sequence.
            Follow the "Marker Sending Rule."
        end
    else
        Record the state of C as the sequence of messages received
        along C after Q's state was recorded and before Q received
        the marker along C.
```

The role of markers in conjunction with FIFO channels is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition given by Eq. 5.9. A marker delineates messages into those that need to be included in the recorded state and those that are not to be recorded in the state. The global state recording algorithm can be initiated by any process by executing the marker sending rule. Also, the global state recording

algorithm can be initiated by several processes concurrently with each process getting its own version of a consistent global state. Each initiation needs its own unique marker (such as <process-id, sequence number>) and different initiations by a process can be distinguished by a local sequence number. A simple way to collect all the recorded information is for each process to send the information it recorded to the initiator of the recording process. The identification of the initiator process can be easily carried by the marker.

A Note on the Collected Global State

It is possible that the global state recorded by the above algorithm is not identical to any of the global states the system actually went through during the computation. This can happen because a site can change its state asynchronously before the markers sent by it are received by other sites. If the state changes while the markers are in transit, the composite of all the states recorded will not correspond to the state of the system at any instant of time. The question of the significance of the recorded global state if the system may have never passed through it arises. Before discussing the utility of a collected global state, we state a result from [8] without giving its proof. This result gives an important property of a collected global state.

Suppose the algorithm is initiated when the system is in global state S_i and it terminates when the system is in global state S_t. Let S_c denote the collected global state by the algorithm, and Seq denote the sequence of actions which take the system from state S_i to S_t. Then, there exists a sequence Seq' that is a permutation of Seq such that S_c can be reached from S_i by executing a prefix of Seq' and S_t can be reached from S_c by executing the rest of the actions in Seq' (see Fig. 5.9).

The usefulness of the recorded global state lies in its ability to detect *stable properties* (a stable property is one that persists) such as the termination of a computation and a deadlock among processes. Note that if a stable property holds before the recording algorithm begins execution, it continues to hold (unless resolved in the case of a deadlock), and will therefore be included in the recorded global state.

Even though the global state recording algorithm can be used for termination detection, it is an expensive way of doing it. In Sec. 5.8 , we give an efficient algorithm for termination detection.

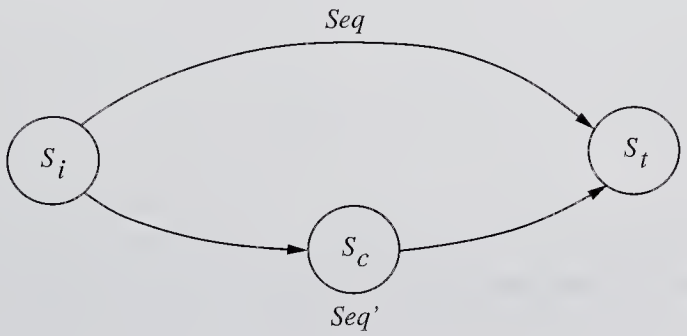


FIGURE 5.9
A property of a collected global state.

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

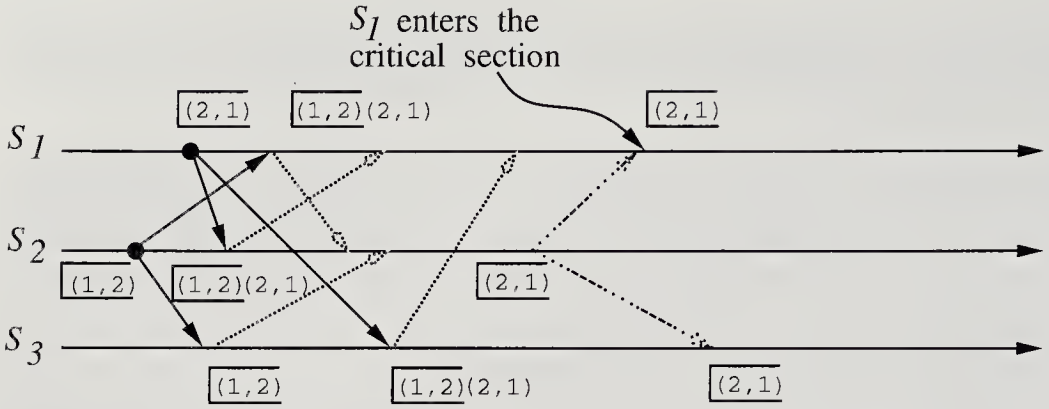


FIGURE 6.6
Site S_1 enters the CS.

in certain situations. For example, suppose site S_j receives a REQUEST message from site S_i after it has sent its own REQUEST message with timestamp higher than the timestamp of site S_i's request. In this case, site S_j need not send a REPLY message to site S_i. This is because when site S_i receives site S_j's request with a timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request that is still pending (because the communication medium preserves message ordering).

6.7 THE RICART-AGRAWALA ALGORITHM

The Ricart-Agrawala algorithm [16] is an optimization of Lamport's algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. In this algorithm also, $\forall i : 1 \leq i \leq N :: R_i = \{S_1, S_2, \ldots, S_N\}$.

The Algorithm

Requesting the critical section.

1. When a site S_i wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.
2. When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS or if site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. The request is deferred otherwise.

Executing the critical section

3. Site S_i enters the CS after it has received REPLY messages from all the sites in its request set.

Releasing the critical section

4. When site S_i exits the CS, it sends REPLY messages to all the deferred requests.

A site's REPLY messages are blocked only by sites that are requesting the CS with higher priority (i.e., a smaller timestamp). Thus, when a site sends out REPLY messages to all the deferred requests, the site with the next highest priority request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their timestamps.

CORRECTNESS

Theorem 6.2. The Ricart-Agrawala algorithm achieves mutual exclusion.

Proof: The proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently and S_i's request has a higher priority (i.e., a smaller timestamp) than the request of S_j. Clearly, S_i received S_j's request after it had made its own request. (Otherwise, S_i's request would have lower priority.) Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j's request) before S_i exits the CS. However, this is impossible because S_j's request has lower priority. Therefore, the Ricart-Agrawala algorithm achieves mutual exclusion. □

In the Ricart-Agrawala algorithm, for every requesting pair of sites, the site with higher priority request will always defer the request of the lower priority site. At any time, only the highest priority request succeeds in getting all the needed REPLY messages.

Example 6.2. Figures 6.7 through 6.10 illustrate the operation of the Ricart-Agrawala algorithm. In Fig. 6.7, sites S_1 and S_2 are making requests for the CS, sending out REQUEST messages to other sites. The timestamps of the requests are (2, 1) and (1, 2), respectively. In Fig. 6.8, S_2 has received REPLY messages from all other sites and consequently, it enters the CS. In Fig. 6.9, S_2 exits the CS and sends a REPLY mesage to site S_1. In Fig. 6.10, site S_1 has received REPLY messages from all other sites and enters the CS next.

PERFORMANCE. The Ricart-Agrawala algorithm requires $2(N-1)$ messages per CS execution: $(N-1)$ REQUEST and $(N-1)$ REPLY messages. Synchronization delay in the algorithm is T.

AN OPTIMIZATION. Roucairol and Carvalho [4] proposed an improvement to the Ricart-Agrawala algorithm by observing that once a site S_i has received a REPLY message from a site S_j, the authorization implicit in this message remains valid until S_i

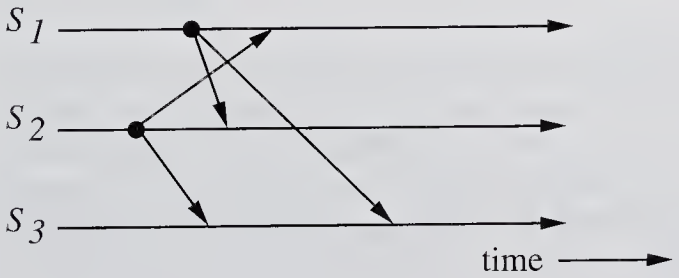


FIGURE 6.7
Sites S_1 and S_2 are making requests for the CS.

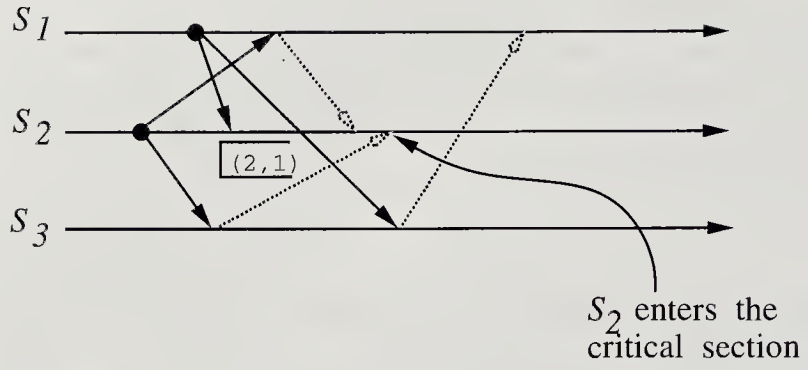


FIGURE 6.8
Site S_2 enter the CS.

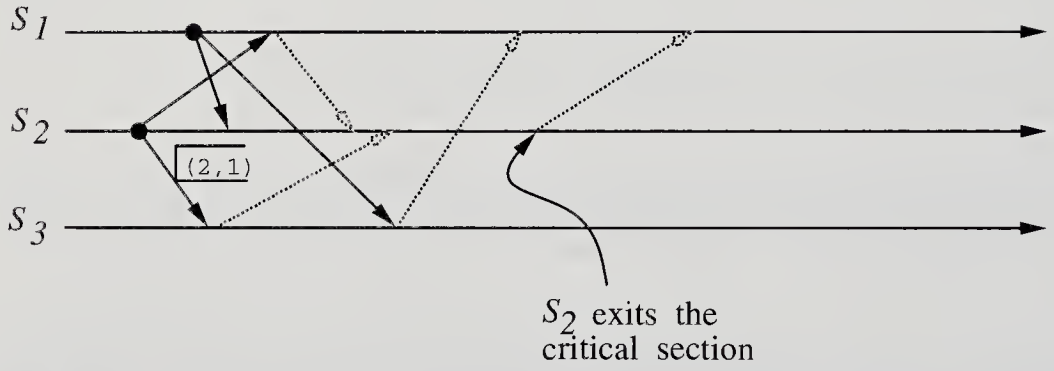


FIGURE 6.9
Site S_2 exits the CS and sends RELEASE messages.

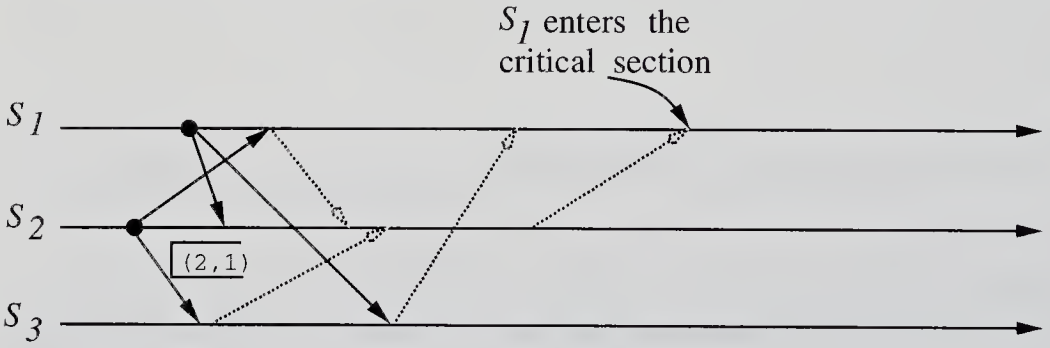


FIGURE 6.10
Site S_1 enters the CS.

sends a REPLY message to S_j (which happens only after the reception of a REQUEST message from S_j). Therefore, after site S_i has received a REPLY message from site S_j, site S_i can enter its CS any number of times without requesting permission from site S_j until S_i sends a REPLY message to S_j. With this change, a site in the Ricart-Agrawala algorithm requests permission from a dynamically varying set of sites and requires 0 to $2(N-1)$ messages per CS execution.

6.9.3 Static vs. Dynamic Information Structures

Non-token-based mutual exclusion algorithms can be classified as either *static* or *dynamic* information structure algorithms. In static information structure algorithms, the contents of request sets, inform sets, and status sets remain fixed and do not change as sites execute CS. Examples of such algorithms are Lamport's [9], Maekawa's [10], and Ricart-Agrawala's [16] algorithms. In dynamic information structure algorithms, the contents of these sets change as the sites execute CS. Examples of such algorithms are found in [4] and [19]. The design of dynamic information structure mutual exclusion algorithms is much more complex because it requires rules for updating the information structure such that the conditions for mutual exclusion are always satisfied. This is the reason that most mutual exclusion algorithms for distributed systems have static information structures.

6.10 TOKEN-BASED ALGORITHMS

In token-based algorithms, a unique token is shared among all sites. A site is allowed to enter its CS if it possesses the token. Depending upon the way a site carries out its search for the token, there are numerous token-based algorithms. Next, we discuss some representative token-based mutual exclusion algorithms.

Before we start with the discussion of token-based algorithms, two comments are in order: First, token-based algorithms use sequence numbers instead of timestamps. Every request for the token contains a sequence number and the sequence numbers of sites advance independently. A site increments its sequence number counter every time it makes a request for the token. A primary function of the sequence numbers is to distinguish between old and current requests. Second, a correctness proof of token-based algorithms to ensure that mutual exclusion is enforced is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS. Rather, the issues of freedom from starvation and freedom from deadlock are prominent.

6.11 SUZUKI-KASAMI'S BROADCAST ALGORITHM

In the Suzuki-Kasami's algorithm [21], if a site attempting to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all the other sites. A site that possesses the token sends it to the requesting site upon receiving its REQUEST message. If a site reeeives a REQUEST message when it is executing the CS, it sends the token only after it has exited the CS. A site holding the token can enter its CS rcpeatedly until it sends the token to some other site.

The main design issues in this algorithm are: (1) distinguishing outdated REQUEST messages from current REQUEST messages and (2) determining which site has an outstanding request for the CS.

Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner: A REQUEST message of site S_j has the form REQUEST(j, n) where n ($n = 1, 2, ...$) is a sequence number that indicates that site S_j is requesting its n^{th} CS execution. A site S_i keeps an array of integers RN_i[1..N] where

PERFORMANCE. A salient feature of this algorithm is that a site can access the critical section without communicating with every site in the system. In low to moderate loads, the average message traffic is $N/2$ because each site sends REQUEST messages to half the sites on average. It increases to N at high loads as most sites will be requesting the CS (which is reflected at site S_i by SV_i[j]= $\mathcal{R}$ for most j's). The synchronization delay in this algorithm is T. An interesting feature of this algorithm is that it adapts itself to the environment of nonuniform traffic of CS requests and to statistical fluctuations in the traffic of CS requests to further reduce the number of messages exchanged.

The algorithm does not have any additional message overhead for the dissemination of state information, except for a slightly larger token message (which is passed comparatively infrequently). Since entries in the token state array (TSV) are either $\mathcal{R}$ or $\mathcal{N}$, this array can be a binary array.

6.13 RAYMOND'S TREE-BASED ALGORITHM

In Raymond's tree-based algorithm [14], sites are logically arranged as a directed tree such that the edges of the tree are assigned directions toward the site (root of the tree) that has the token. Every site has a local variable *holder* that points to an immediate neighbor node on a direeted path to the root node. Thus, *holder* variables at the sites define logical tree structure among the sites. If we follow *holder* variables at sites, every site has a directed path leading to the site holding the token. At root site, *holder* points to itself. An example of a tree configuration is shown in Fig. 6.12.

Every site keeps a FIFO queue, called *request_q*, which stores the requests of those neighboring sites that have sent a request to this site, but have not yet been sent the token.

The Algorithm

Requesting the critical section

1. When a site wants to enter the CS, it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its *request_q*

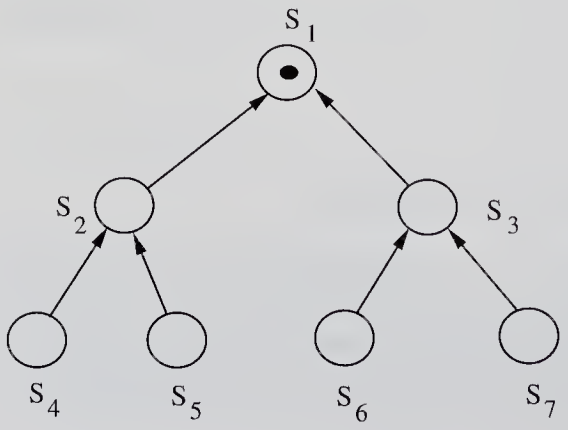


FIGURE 6.12
Sites arranged in a tree configuration.

is empty. It then adds its request to its $request\_q$. (Note that a nonempty $request\_q$ at a site indicates that the site has sent a REQUEST message to the root node for the top entry in its $request\_q$.)

2. When a site on the path receives this message, it places the REQUEST in its $request\_q$ and sends a REQUEST message along the directed path to the root provided it has not sent out a REQUEST message on its outgoing edge (for a previously received REQUEST on its $request\_q$).
3. When the root site receives a REQUEST message, it sends the token to the site from which it received the REQUEST message and sets its $holder$ variable to point at that site.
4. When a site receives the token, it deletes the top entry from its $request\_q$, sends the token to the site indicated in this entry, and sets its $holder$ variable to point at that site. If the $request\_q$ is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by $holder$ variable.

Executing the critical section

5. A site enters the CS when it receives the token and its own entry is at the top of its $request\_q$. In this case, the site deletes the top entry from its $request\_q$ and enters the CS.

Releasing the critical section. After a site has finished execution of the CS, it takes the following actions:

6. If its $request\_q$ is nonempty, then it deletes the top entry from its $request\_q$, sends the token to that site, and sets its $holder$ variable to point at that site.
7. If the $request\_q$ is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by the $holder$ variable.

CORRECTNESS. The algorithm is free from deadlocks because the acyclic nature of tree configuration eliminates the possibility of *circular wait* among requesting sites. We next show that the algorithm is free from starvation.

Theorem 6.7. A requesting site enters the CS in finite time.

Proof: A formal correctness proof is long and complex. Thus, an informal correctness proof is provided. For a formal proof, readers are referred to the original paper [14].

The essence of proof is based on the following two facts: (1) a site serves requests in its $request\_q$ in the FCFS order and (2) every site has a path leading to the site that has the token. Due to the latter fact and Step 2 of the algorithm, when a site S_i is making a request, there exists a chain of requests from site S_i to site S_h, which holds the token. Let the chain be denoted by S_i, S_{i1}, S_{i2}, ..., S_{ik-1}, S_{ik}, S_h. When S_h receives a REQUEST message from S_{ik}, it sends the token to S_{ik}. There are two possibilities: S_{ik-1}'s request is at the top of S_{ik}'s $request\_q$ or it is not at

the top. In the first case, S_{ik} sends the token to site S_{ik-1}. In the second case, S_{ik} sends the token to the site, say S_j, at the top of its $request\_q$ and also sends it a REQUEST message (see Step 4 of the algorithm). This extends the chain of requests to S_i, S_{i1}, S_{i2}, ..., S_{ik-1}, S_{ik}, S_j, ..., S_l, where site S_l executes the CS next. Note that due to fact (1) above, all the sites in the chain S_j, ..., S_l will execute the CS at most once before the token is returned to site S_{ik}. Thus, site S_{ik} sends the token to S_{ik-1} in finite time. Likewise, S_{ik-1} sends the token to S_{ik-2} in finite time and so on. Eventually, S_{i1} sends the token to S_i. Consequently, a requesting site eventually receives the token. □

Example 6.4. In Fig. 6.13, site S_5 sends a REQUEST message to S_2, which propagates it to the root S_1. Root S_1 sends the token to S_2 which in turn sends the token to S_5 (Fig. 6.14). The token travels along the same path traveled by the REQUEST message (but in the opposite direction) and it also reverses the direction of the edges on the path. Consequently, the site that executes the CS last becomes the new root. (See Fig. 6.15.)

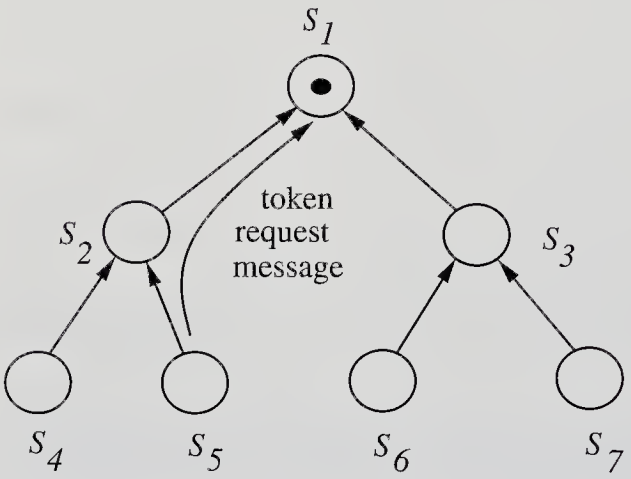


FIGURE 6.13
Site S_5 is requesting the token.

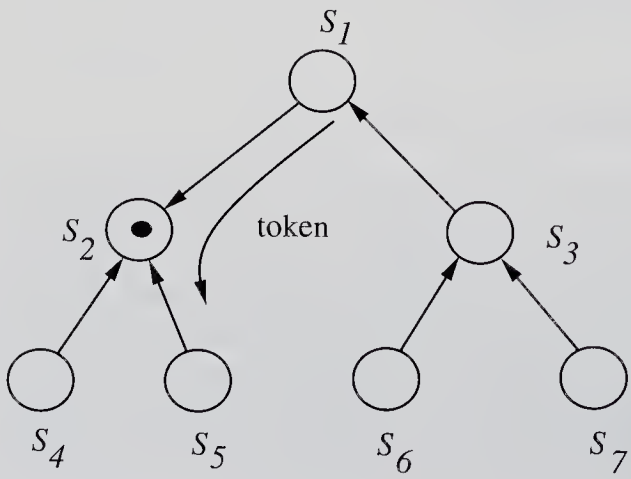


FIGURE 6.14
The token is in transit to S_5.

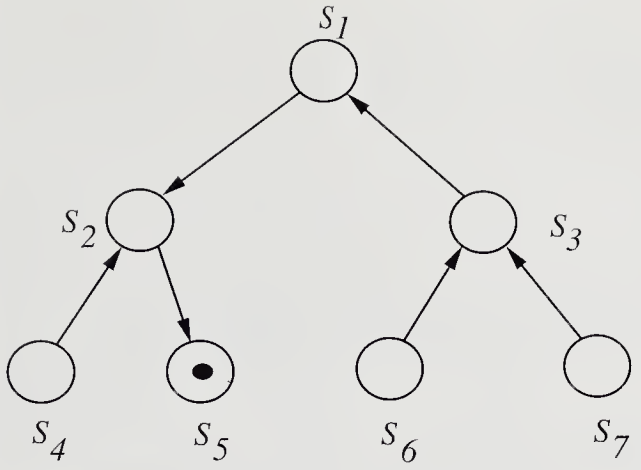


FIGURE 6.15
State after S_5 has received the token.

PERFORMANCE. The average message complexity of Raymond's algorithm is $O(\log N)$ because the average distance between any two nodes in a tree with N nodes is $O(\log N)$. Synchronization delay in this algorithm is $(T \log N)/2$ because the average distance between two sites to successively execute the CS is $(\log N)/2$.

Raymond's algorithm can use *greedy strategy*, where a site receiving the token executes the CS even though its request is not at the top of its $request\_q$. It is important to note that in heavy loads, the synchronization delay in this case becomes T because a site executes the CS every time the token is transferred. Needless to say, the greedy strategy has an adverse effect on the fairness of the algorithm and can cause starvation.

6.14 A COMPARATIVE PERFORMANCE ANALYSIS

In this section, we present a performance comparison among various mutual exelusion algorithms. Table 6.1 summarizes the response time, the number of messages required, and synchronization delay for mutual exclusion algorithms discussed in this chapter.

6.14.1 Response Time

At low loads, there is hardly any eontention among requests for the CS. Therefore, the response time under low load conditions for many algorithms is simply a round trip message delay $(= 2T)$ to acquire permission or token plus the time to execute the CS $(= E)$. In Raymond's algorithm, the average distance between a requesting site and the site holding the token is $(\log N)/2$. Thus, the average round trip delay to acquire the token is $T(\log N)$.

As the load is increased, response time inereases in all mutual exclusion algorithms beeause contention for access to the CS increases. Different algorithms see different increases in response time with respect to load. A closed form expression of response time as a funetion of load is not known for these algorithms. The response time under heavy load conitions is diseussed in See. 6.14.4 (see Table 6.2 under "Maximum average response time").

Several problems of this algorithm (such as large response time and the congestion of communication links near the control site) can be mitigated by having each site maintain its resource status (WFG) locally and by having each site send its resource status to a designated site periodically for construction of the global WFG and the detection of deadlocks [11]. However, due to inherent communication delays and the lack of perfectly synchronized clocks, the designated site may get an inconsistent view of the system and detect false deadlocks [14].

For example, suppose two resources R_1 and R_2 are stored at sites S_1 and S_2, respectively. Suppose the following two transactions T_1 and T_2 are started almost simultaneously at sites S_3 and S_4, respectively:

T_1	T_2
lock R_1	lock R_1
unlock R_1	unlock R_1
lock R_2	lock R_2
unlock R_2	unlock R_2

Suppose that the lock(R_1) request of T_1 arrives at S_1 and locks R_1 followed by the lock(R_1) request of T_2, which waits at S_1. At this point S_1 reports its status, $T_2 \rightarrow T_1$ to a designated site. Thereafter, T_1 unlocks R_1, T_2 locks R_1, T_1 makes a loek(R_2) request to S_2, T_2 unlocks R_1 and makes a lock(R_2) request to S_2. Now suppose that the lock(R_2) request of T_2 arrives at S_2 and locks R_2 followed by the loek(R_2) request of T_1 which waits at S_2. At this point S_2 reports its status, $T_1 \rightarrow T_2$ to the designated site, which after constructing the global WFG, reports a false deadlock $T_1 \rightarrow T_2 \rightarrow T_1$.

7.6.2 The Ho-Ramamoorthy Algorithms

Ho and Ramamoorthy gave two centralized deadlock detection algorithms, ealled two-phase and one-phase algorithms [14], to fix the problem of the above algorithm. These algorithms, respectively, collect two consecutive status reports or keep two status tables at each site to ensure that the control site gets a consistent view of the system.

THE TWO-PHASE ALGORITHM. In the two-phase algorithm, every site maintains a status table that contains the status of all the processes initiated at that site. The status of a process includes all resources locked and all resources being waited upon. Periodically, a designated site requests the status table from all sites, constructs a WFG from the information received, and searches it for cycles. If there is no cycle, then the system is free from deadlocks, otherwise, the designated site again requests status tables from all the sites and again constructs a WFG using *only* those transactions which are common to both reports. If the same cycle is detected again, the system is declared deadlocked.

It was claimed that by selecting only the eommon transactions found in two consecutive reports, the algorithm gets a consistent view of the system. (A view is consistent if it reflects a correct state of the system.) If a deadlock exists, it was argued,

the same wait-for condition must exist in both reports. However, this claim proved to be incorrect (i.e., a cycle in the wait-for conditions of the transactions common in two consecutive reports does not imply a deadlock) and two-phase algorithm may indeed report false deadlocks. By getting two consecutive reports, the designated site reduces the probability of getting an inconsistent view, but does not eliminate such a possibility.

THE ONE-PHASE ALGORITHM. The one-phase algorithm requires only one status report from each site; however, each site maintains two status tables: a *resource status* table and a *process status* table. The resource status table at a site keeps track of the transactions that have locked or are waiting for resources stored at that site. The process status table at a site keeps track of the resources locked by or waited for by all the transactions at that site. Periodically, a designated site requests both the tables from every site, constructs a WFG using only those transactions for which the entry in the resource table matches the corresponding entry in the process table, and searches the WFG for cycles. If no cycle is found, then the system is not deadlocked, otherwise a deadlock is detected.

The one-phase algorithm does not detect false deadlocks because it eliminates the inconsistency in state information by using only the information that is common to both tables. This eliminates inconsistencies introduced by unpredictable message delays. For example, if the resource table at site S_1 indicates that resource R_1 is waited upon by a process P_2 (i.e., $R_1 \leftarrow P_2$) and the process table at site S_2 indicates that process P_2 is waiting for resource R_1 (i.e., $P_2 \rightarrow R_1$), then edge $P_2 \rightarrow R_1$ in the constructed WFG reflects the correct system state. If either of these entries is missing from the resource or the process table, then a request message or a release message from S_2 to S_1 is in transit and $P_2 \rightarrow R_1$ cannot be ascertained.

The one-phase algorithm is faster and requires fewer messages as compared to the two-phase algorithm. However, it requires more storage because every site maintains two status tables and exchanges bigger messages because a message contains two tables instead of one.

7.7 DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

In distributed deadlock detection algorithms, all sites collectively cooperate to detect a cycle in the state graph that is likely to be distributed over several sites of the system. A distributed deadlock detection algorithm can be initiated whenever a process is forced to wait, and it can be initiated either by the local site of the process or by the site where the process waits.

Distributed deadlock detection algorithms can be divided into four classes [18]: *path-pushing, edge-chasing, diffusion computation,* and *global state detection.*

In path-pushing algorithms, the wait-for dependency information of the global WFG is disseminated in the form of paths (i.e., a sequence of wait-for dependency edges). Classic examples of such algorithms are Menasce-Muntz [22] and Obermarck [24] algorithms.

Obermarck's algorithm has two interesting features:

- The nonlocal portion of the global TWF graph at a site is abstracted by a distinguished node (called External or Ex) which helps in determining potential multisite deadlocks without requiring a huge global TWF graph to be stored at each site.
- Transactions are totally ordered, which reduces the number of messages and consequently decreases deadlock detection overhead. It also ensures that exactly one transaction in each cycle detects the deadlock.

THE ALGORITHM. Deadlock detection at a site follows the following iterative process:

1. The site waits for deadlock-related information (produced in Step 3 of the previous deadlock detection iteration) from other sites. (Note that deadlock-related information is passed by sites in the form of paths.)
2. The site combines the received information with its local TWF graph to build an updated TWF graph. It then detects all cycles and breaks only those cycles which do not contain the node 'Ex'. Note that these cycles are local to this site. All other cycles have the potential to be a part of global cycles.
3. For all cycles 'Ex $\rightarrow T_1 \rightarrow T_2 \rightarrow$ Ex' which contain the node 'Ex' (these cycles are potential candidates for global deadlocks), the site transmits them in string form 'Ex, T_1, T_2, Ex' to all other sites where a subtransaction of T_2 is waiting to receive a message from the subtransaction of T_2 at this site. The algorithm reduces message traffic by lexically ordering transactions and sending the string 'Ex, T_1, T_2, T_3, Ex' to other sites only if T_1 is higher than T_3 in the lexical ordering. Also, for a deadlock, the highest priority transaction detects the deadlock.

Obermarck gave an informal correctness proof of the algorithm [24]. However, the algorithm is incorrect because it detects phantom deadlocks. The main reason for this is that the portions of TWF graphs that are propagated to other sites may not represent a consistent view of the global TWF graph. This is because each site takes its snapshot asynchronously at Step 2. Consequently, when a site sends out portions of its TWF graph as paths to other sites in Step 3, the global dependency represented by this path may change without this site knowing about it.

This algorithm sends $n(n-1)/2$ messages to detect a deadlock involving n sites. Size of a message is $O(n)$. The delay in detecting the deadlock is $O(n)$.

7.7.2 An Edge-Chasing Algorithm

We discuss Chandy-Misra-Haas's distributed deadlock detection algorithm [5] for the AND request model to illustrate deadlock detection using edge-chasing.

Chandy et al.'s algorithm [5] uses a special message called a probe. A *probe* is a triplet (i, j, k) denoting that it belongs to a deadlock detection initiated for process P_i and it is being sent by the home site of process P_j to the home site of process P_k.

A probe message travels along the edges of the global TWF graph, and a deadlock is detected when a probe message returns to its initiating process.

We now define terms and data structures used in the algorithm. A process P_j is said to be *dependent* on another process P_k if there exists a sequence of processes P_j, P_{i1}, P_{i2}, ..., P_{im}, P_k such that each process except P_k in the sequence is blocked and each process, except the first one (P_j), holds a resource for which the previous process in the sequence is waiting. Process P_j is *locally dependent* upon process P_k if P_j is dependent upon P_k and both the processes are at the same site. The system maintains a boolean array, $dependent_i$, for each process P_i, where $dependent_i(j)$ is true only if P_i knows that P_j is dependent on it. Initially, $dependent_i(j)$ is false for all i and j.

THE ALGORITHM. To determine if a blocked process is deadlocked, the system executes the following algorithm:

if P_i is locally dependent on itself
then declare a deadlock
else for all P_j and P_k such that
(a) P_i is locally dependent upon P_j, and
(b) P_j is waiting on P_k, and
(c) P_j and P_k are on different sites,
send probe (i, j, k) to the home site of P_k

On the receipt of probe (i, j, k), the site takes the following actions:

if
(d) P_k is blocked, and
(e) $dependent_k(i)$ is false, and
(f) P_k has not replied to all requests of P_j,
then
begin
$dependent_k(i)$ = true;
if $k = i$
then declare that P_i is deadlocked
else for all P_m and P_n such that
(a') P_k is locally dependent upon P_m, and
(b') P_m is waiting on P_n, and
(c') P_m and P_n are on different sites,
send probe (i, m, n) to the home site of P_n
end.

Thus, a probe message is successively propagated along the edges of the global TWF graph and a deadlock is detected when a probe message returns to its initiating process.

Example 7.1. As an example, consider the system shown in Fig. 7.1. If process P_1 initiates deadlock detection, it sends probe (1, 3, 4) to S_2. Since P_6 is waiting for P_8 and P_7 is waiting for P_{10}, S_2 sends probes (1, 6, 8) and (1, 7, 10) to S_3 which in

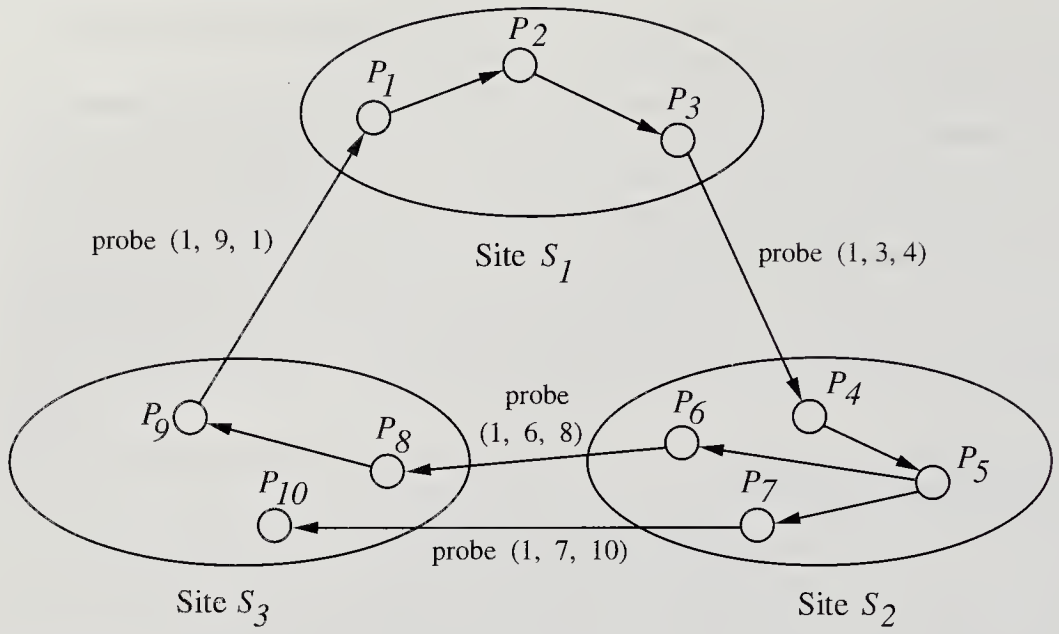


FIGURE 7.1
An example of Chandy et al.'s edge-chasing algorithm.

turn sends probe (1, 9, 1) to S_1. On receiving probe (1, 9, 1), S_1 declares that P_1 is deadlocked.

Chandy et al.'s algorithm sends one probe message (per deadlock detection initiation) on each edge of the WFG, which spans two sites. Thus, the algorithm at most exchanges $m(n-1)/2$ messages to deteet a deadlock that involves m processes and spans over n sites. The size of messages exchanged is fixed and very small (only 3 integer words). The delay in detecting the deadlock is $O(n)$.

OTHER EDGE-CHASING ALGORITHMS

The Mitchell-Merritt Algorithm. In the deadlock detection algorithm of Mitchell and Merritt [23], each node of the TWF graph has two labels: private and public. The private label of each node is unique to that node, and initially both labels at a node have the same value. The algorithm detects a deadlock by propagating the public label of nodes in the backward direction in the TWF graph. When a transaction is blocked, the public and private label of its node in the TWF graph are changed to a value greater than their previous values and greater than the public label of the blocking transaction. A blocked transaction periodically reads the public label of the blocking transaction and replaces its own public label with it, provided the blocking transaction's public label is larger than its own. A deadlock is detected when a transaction receives its own public label. In essence, the largest public label propagates in the backward direction in a deadlock cycle. Deadlock resolution is simple in this algorithm because only one process detects a deadlock and that process can resolve the deadlock by simply aborting itself.

Sinha-Natarajan Algorithm. In the Sinha-Natarajan algorithm [28], transactions are assigned unique priorities, and an *antagonistic* conflict is said to occur when a transaction waits for a data object that is locked by a lower priority transaction. The algorithm initiates deadlock detection only when an antagonistic conflict occurs.

The algorithm detects a deadlock by circulating a probe message through a cycle in the global TWF graph. A probe message is a 2-tuple (i, j) where T_i is the transaction that initiated the deadlock detection and T_j is the transaction whose priority is the lowest among all the transactions (i.e., nodes of the TWF graph) the probe has traversed so far. When a waiting transaction receives a probe that was initiated by a lower priority transaction, the probe is discarded.

An interesting property of this algorithm is that a deadlock is detected when the probe issued by the highest priority transaction in the cycle returns to it. (There is only one detector of every deadlock.) Deadlock resolution is simple because the detector of a deadlock can resolve the deadlock by aborting the lowest priority transaction of the cycle. This was the first algorithm to comprehensively treat deadlock resolution.

Choudhary et al. showed that the Sinha-Natarajan algorithm detects false deadlocks and fails to report all deadlocks because it overlooks the possibility of a transaction waiting transitively on a deadlock cycle and because the probes of aborted transactions are not cleaned properly [7]. Choudhary et al. proposed a corrected version of the Sinha-Natarajan algorithm, but it has been shown that Choudhary et al.'s corrected algorithm still detects false deadlocks and fails to report all deadlocks [19].

7.7.3 A Diffusion Computation Based Algorithm

In diffusion computation based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system. Chandy et al.'s distributed deadlock detection algorithm for the OR request model [5] is discussed to illustrate the technique of diffusion computation based algorithms.

A process determines if it is deadlocked by initiating a diffusion computation. The messages used in diffusion computation take the form of a query(i, j, k) and a reply(i, j, k), denoting that they belong to a diffusion computation initiated by a process P_i and are being sent from process P_j to process P_k. A process can be in two states: active or blocked. In the active state, a process is executing and in the blocked state, a process is waiting to acquire a resource. A blocked process initiates deadlock detection by sending query messages to all the processes from whom it is waiting to receive a message (these processes are called the *dependent set* of the process).

If an active process receives a query or reply message, it discards it. When a blocked process P_k receives a query(i, j, k) message, it takes the following actions:

- If this is the first query message received by P_k for the deadlock detection initiated by P_i (called the *engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable $num_k(i)$ to the number of query messages sent.
- If this is not an engaging query, then P_k returns a reply message to it immediately, provided P_k has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.

A local boolean variable $wait_k(i)$ at process P_k denotes the fact that it has been continuously blocked since it received the last engaging query from process P_i. When a blocked process P_k receives a reply(i, j, k) message, it decrements $num_k(i)$ provided $wait_k(i)$ holds. A process sends a reply message in response to an engaging query only after it has received a reply to every query message it sent out for this engaging query; i.e., $num(i) = 0$ at it.

An initiator detects a deadlock when it receives reply messages to all the query messages it had sent out.

THE ALGORITHM. We now describe the Chandy et al.'s diffusion computation based deadlock detection algorithm in pseudocode for the OR request model [5], [18]:

Initiate a diffusion computation for a blocked process P_i:

send query$(i,\ i,\ j)$ to all processes P_j in the dependent set DS_i of P_i;
$num_i(i)$:= $|DS_i|$; $wait_i(i)$:= true;

When a blocked process P_k receives a query(i, j, k):

if this is the engaging query for process P_k
then send query$(i,\ k,\ m)$ to all P_m in its dependent set DS_k;
$num_k(i)$:= $|DS_k|$; $wait_k(i)$:= true
else if $wait_k(i)$ then send a $reply(i,\ k,\ j)$ to P_j.

When a process P_k receives a reply(i, j, k)

if $wait_k(i)$
then begin
$num_k(i)$:= $num_k(i) - 1$;
if $num_k(i) = 0$
then if $i = k$ then **declare a deadlock**
else send reply$(i,\ k,\ m)$ to the process P_m,
which sent the engaging query.

In the above description of the algorithm, we assumed that only one diffusion computation is initiated for a process. In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process is blocked). However, note that at any time only one diffusion computation is current for any process. All others are outdated. The current diffusion computation can be distinguished from outdated ones by using sequence numbers (see [5]).

7.7.4 A Global State Detection Based Algorithm

There are three deadlock detection algorithms to detect generalized distributed deadlocks using global state detection approach. The algorithm by Bracha and Toueg [2] consists

of two phases. In the first phase, the algorithm records a snapshot of a distributed WFG and in the second phase, the algorithm simulates the granting of requests to check for generalized deadlocks. The second phase is nested within the first phase. Therefore, the first phase terminates after the second phase has terminated. The algorithm by Wang et al. [30] also consists of two phases. In the first phase, the algorithm records a snapshot of the distributed WFG. In the second phase, the static WFG recorded in the first phase is reduced to detect any deadlocks. Both the phases occur serially.

The Kshemkalyani-Singhal algorithm [20] has a single phase, which consists of a fan-out sweep of messages outwards from an initiator process and a fan-in sweep of messages inwards to the initiator process. A *sweep* of a WFG is a traversal of the WFG in which all messages are sent in the direction of the WFG edges (an outward sweep) or all messages are sent against the direction of the WFG edges (an inward sweep). Both the outward and the inward sweeps are done concurrently in the algorithm. In the outward sweep, the algorithm records a snapshot of a distributed WFG. In the inward sweep, the recorded distributed WFG is reduced to determine whether the initiator is deadlocked. This algorithm deals with the complications introduced because the two sweeps can overlap in time at any process, i.e., the reduction of the WFG at a process can begin before all the WFG edges incident at that process have been reeorded.

SYSTEM MODEL. The system has n nodes, with every node connected to every other node by a logical channel. An event in a computation ean be an internal event, a message send event, or a message receive event. Events are assigned timestamps as per Lamport's clock scheme [21]. The timestamp of an event that occurs at time t on node i is denoted t_i.

The computation messages can either be REQUEST, REPLY or CANCEL messages. A node i sends q_i REQUESTs to q_i other nodes when it blocks (goes from an active to a blocked state) on a p_i-out-of-q_i request. When node i blocks on node j node j becomes a successor of node i and node i becomes a predecessor of node j in the WFG. A REPLY message denotes the granting of a request. A node i unblocks when p_i out of its q_i requests are granted. When the node unblocks, it sends CANCEL messages to withdraw the remaining $q_i - p_i$ requests it had sent.

The sending and receiving of REQUEST, REPLY, and CANCEL messages are *computation events*. The sending and receiving of deadlock detection algorithm messages are *algorithm events*.

A node i has the following local variables to record its state:

$wait_i$: boolean (:= $false$); /*records the current status.*/
t_i: integer (:= 0); /*current time.*/
$in(i)$: set of nodes whose requests are outstanding at i
$out(i)$: set of nodes on which i is waiting.
p_i: integer (:= 0); /*the number of replies required for unblocking.*/
w_i: real (:= 1.0); /*weight to detect termination of deadlock detection algorithm.*/

REQUEST_SEND(i).

/*Executed by node i when it blocks on a p_i-out-of-q_i request.*/
For every node j on which i is blocked do
    $out(i) \leftarrow out(i) \bigcup \{j\}$;
    send REQUEST(i) to j;
set p_i to the number of replies needed;
$wait_i \leftarrow true$;

REQUEST_RECEIVE(j).

/*Executed by node i when it receives a request made by j */
$in(i) \leftarrow in(i) \bigcup \{j\}$.

REPLY_SEND(j).

/*Executed by node i when it replies to a request by j.*/
$in(i) \leftarrow in(i) - \{j\}$;
send REPLY(i) to j.

REPLY_RECEIVE(j).

/*Executed by node i when it receives a reply from j to its request.*/
if valid reply for the current request
    then begin
    $out(i) \leftarrow out(i) - \{j\}$;
    $p_i \leftarrow p_i - 1$;
    $p_i = 0 \rightarrow$
        $\{wait_i \leftarrow false$;
        $\forall\ k \in out(i)$, **send** CANCEL(i) to k;
        $out(i) \leftarrow \emptyset.\}$
    end

CANCEL_RECEIVE(j).

/*Executed by node i when it receives a cancel from j.*/
if $j \in in(i)$ then $in(i) \leftarrow in(i) - \{j\}$.

When a node $init$ blocks on a P-out-of-Q request, it initiates the deadlock detection algorithm. The algorithm records part of the WFG that is reachable from $init$ (henceforth, referred to as $init$'s WFG) in a distributed snapshot [4]; such a distributed snapshot includes only those dependency edges and nodes that form $init$'s WFG. When multiple nodes block concurrently, they may each initiate the deadlock detection algorithm concurrently. Each invocation of the deadlock detection algorithm is treated independently and is identified by the initiator's identity and initiator's timestamp when it is blocked. Every node maintains a local snapshot for the latest deadlock detection algorithm initiated by every other node. We will describe only a single instance of the deadlock detection algorithm.

AN INFORMAL DESCRIPTION OF THE ALGORITHM. The distributed WFG is recorded using FLOOD messages in the outward sweep and is examined for deadlocks using ECHO messages in the inward sweep. To detect a deadlock, the initiator *init* records its local state and sends FLOOD messages along its outward dependencies when it blocks. When node i receives the first FLOOD message along an existing inward dependency, it records its local state. If node i is blocked at this time, it sends out FLOOD messages along its outward dependencies to continue the recording of the WFG in the outward sweep. If node i is active at this time, (i.e., it does not have any outward dependencies and is a leaf node in the WFG), then it initiates reduction of the WFG by returning an ECHO message along the incoming dependency even before the states of all incoming dependencies have been recorded in the WFG snapshot at the leaf node.

ECHO messages perform the reduction of the recorded WFG by simulating the granting of requests in the inward sweep. A node i in the WFG is reduced if it receives ECHOs along p_i out of its q_i outgoing edges indicating that p_i of its requests can be granted. An edge is reduced if an ECHO is received on the edge indicating that the request it represents can be granted. After a local snapshot has been recorded at node i, any transition made by i from an idle to an active state is captured in the process of reduction. The nodes that can be reduced do not form a deadlock whereas the nodes that cannot be reduced are deadlocked. The order in which the reduction of the nodes and edges of the WFG is performed does not alter the final result. Node *init* detects the deadlock if it is not reduced when the deadlock detection algorithm terminates.

In general, WFG reduction can begin at a nonleaf node before the recording of the WFG has been completed at that node. This happens when an ECHO message arrives and begins reduction at a nonleaf node before all the FLOODs have arrived and recorded the complete local WFG at that node. Thus, the activities of recording and reducing the WFG snapshot are done concurrently in a single phase and no serialization is imposed between the two activities as is done in [30]. Since a reduction is done on an incompletely recorded WFG at the nodes, the local snapshot at each node has to be carefully manipulated so as to give the effect that WFG reduction is initiated after WFG recording has been completed.

TERMINATION DETECTION. A termination detection technique based on weights [16] detects the termination of the algorithm using SHORT messages (in addition to FLOODs and ECHOs). A weight of 1.0 at the initiator node, when the algorithm is initiated, is distributed among all FLOOD messages sent out by the initiator. When the first FLOOD is received at a nonleaf node, the weight of the received FLOOD is distributed among the FLOODs sent out along outward edges at that node to expand the WFG further. Since any subsequent FLOOD arriving at a nonleaf node does not expand the WFG further, its weight is returned to the initiator through a SHORT message. When a FLOOD is received at a leaf node, its weight is transferred to the ECHO sent by the leaf node to reduce the WFG. When an ECHO that arrives at a node unblocks the node, the weight of the ECHO is distributed among the ECHOs that are sent by that node along the incoming edges in its WFG snapshot. When an ECHO arriving at a node does not unblock the node, its weight is sent directly to the initiator through a SHORT message.

The algorithm maintains the invariant such that the sum of the weights in FLOOD, ECHO, and SHORT messages plus the weight at the initiator (received in SHORT and ECHO messages) is always one. The algorithm terminates when the weight at the initiator becomes 1.0, signifying that all WFG recording and reduction activity has completed.

THE ALGORITHM. $FLOOD, ECHO$, and $SHORT$ control messages use weights (introduced in [16]) for termination detection. The weight w is a real number in the range $[0, 1]$.

A node i stores the local snapshot for snapshots *initiated* by every other node to detect deadlock in a data structure LS, which is an array of records.

LS: array [1..N] of record;

A record has several fields to record snapshot related information and is defined below for initiator $init$:

$LS[init].out$: set of integers (:= $\emptyset$); /* nodes on which i is waiting in the snapshot. */
$LS[init].in$: set of integers (:= $\emptyset$); /* nodes waiting on i in the snapshot */
$LS[init].t$: integer (:= 0); /* time when $init$ initiated snapshot. */
$LS[init].s$: boolean (:= $false$); /* local blocked state as seen by snapshot. */
$LS[init].p$: integer; /* value of p_i as seen in snapshot. */

The deadlock detection algorithm is defined by the following procedures.

SNAPSHOT_INITIATE.

/* Executed by node i to detect whether it is deadlocked. */
$init \leftarrow i$;
$w_i \leftarrow 0$;
$LS[init].t \leftarrow t_i$;
$LS[init].out \leftarrow out(i)$;
$LS[init].s \leftarrow true$;
$LS[init].in \leftarrow \emptyset$;
$LS[init].p \leftarrow p_i$;
send $FLOOD(i, i, t_i, 1/|out(i)|)$ to each j in $out(i)$. /* $1/|out(i)|$ is the fraction of weight sent in a FLOOD message. */

FLOOD_RECEIVE(j, $init$, t_init, w).

/* Executed by node i on receiving a FLOOD message from j. */
[
$LS[init].t < t\_init \bigwedge j \in in(i) \rightarrow$ /* Valid FLOOD for a new snapshot. */
$LS[init].out \leftarrow out(i)$;
$LS[init].in \leftarrow \{j\}$;
$LS[init].t \leftarrow t\_init$;
$LS[init].s \leftarrow wait_i$;

$wait_i = true \rightarrow$
$LS[init].p \leftarrow p_i$;
send $FLOOD(i, init, t\_init, w/|out(i)|)$ to each $k \in out(i)$;
$wait_i = false \rightarrow$
$LS[init].p \leftarrow 0$
send $ECHO(i, init, t\_init, w)$ to j;
$LS[init].in \leftarrow LS[init].in - \{j\}$.

□
$LS[init].t < t\_init \bigwedge j \notin in(i) \rightarrow$ /* Invalid FLOOD for a new */
send $ECHO\ (i, init, t\_init, w)$ to j. /* snapshot. */

□
$LS[init].t = t\_init \bigwedge j \notin in(i) \rightarrow$ /* Invalid FLOOD for a current */
send $ECHO\ (i, init, t\_init, w)$ to j. /* snapshot.*/

□
$LS[init].t = t\_init \bigwedge j \in in(i) \rightarrow$ /* Valid FLOOD for a current */
$LS[init].s = false \rightarrow$ /* snapshot.*/
send $ECHO(i, init, t\_init, w)$ to j;
$LS[init].s = true \rightarrow$
$LS[init].in \leftarrow LS[init].in \bigcup \{j\}$;
send $SHORT(init, t\_init, w)$ to $init$.

□
$LS[init].t > t\_init \rightarrow$ discard the FLOOD message. /*Out-dated FLOOD. */
]

ECHO_RECEIVE(j, $init$, t_init, w).

/*Executed by node i on receiving an ECHO from j. */
[
/*Echo for out-dated snapshot. */
$LS[init].t > t\_init \rightarrow$ discard the ECHO message.

□
$LS[init].t < t\_init \rightarrow$ cannot happen. /*ECHO for unseen snapshot. */

□
$LS[init].t = t\_init \rightarrow$ /*ECHO for current snapshot. */
$LS[init].out \leftarrow LS[init].out - \{j\}$;
$LS[init].s = false \rightarrow$ **send** $SHORT(init, t\_init, w)$ to $init$.
$LS[init].s = true \rightarrow$
$LS[init].p \leftarrow LS[init].p - 1$;
$LS[init].p = 0 \rightarrow$ /* getting reduced */
$LS[init].s \leftarrow$ false;
$init = i \rightarrow$ declare not deadlocked; exit.
send $ECHO(i, init, t\_init, w/|LS[init].in|)$
to all $k \in LS[init].in$;
$LS[init].p \neq 0 \rightarrow$
send $SHORT(init, t\_init, w)$ to $init$.
]

SHORT_RECEIVE($init$, t_init, w)

/*Executed by node i (which is always $init$) on receiving a SHORT. */
[
/*SHORT for out-dated snapshot. */
$t\_init < t\_block_i \rightarrow$ discard the message.
□
/*SHORT for uninitiated snapshot. */
$t\_init > t\_block_i \rightarrow$ not possible.
□
/*SHORT for currently initiated snapshot. */
$t\_init = t\_block_i \bigwedge LS[init].s = false \rightarrow$ discard.
$t\_init = t\_block_i \bigwedge LS[init].s = true \rightarrow$
$w_i \leftarrow w_i + w$;
$w_i = 1 \rightarrow$ **declare deadlock and abort**.
]

The algorithm has a message complexity of $4e - 2n + 2l$ and a time complexity of $2d$ hops, where e is the number of edges, n the number of nodes, l the number of leaf nodes, and d the diameter of the WFG. This is better than the two-phase algorithms of Bracha and Toueg [2] and Wang et al. [30] and gives the best time complexity of any algorithm that reduces a distributed WFG to detect generalized distributed deadlocks.

7.8 HIERARCHICAL DEADLOCK DETECTION ALGORITHMS

In hierarchical algorithms, sites are (logically) arranged in hierarchical fashion, and a site is responsible for detecting deadlocks involving only its children sites. These algorithms take advantage of access patterns that are localized to a cluster of sites to optimize performance.

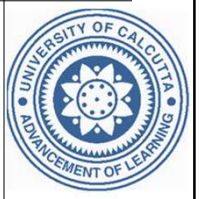
7.8.1 The Menasce-Muntz Algorithm

In the hierarchical deadlock detection algorithm of Menasce and Muntz [22], all the controllers are arranged in tree fashion. (A *controller* manages a resource or is responsible for deadlock detection.) The controllers at the bottom-most level (called *leaf controllers*) manage resources and others (called *nonleaf controllers*) are responsible for deadlock detection. A leaf controller maintains a part of the global TWF graph concerned with the allocation of the resources at that leaf controller. A nonleaf controller maintains all TWF graphs spanning its children controllers and is responsible for detecting all deadlocks involving all of its leaf controllers.

Whenever a change occurs in a controller's TWF graph due to a resource allocation, wait, or release, it is propagated to its parent controller. The parent controller makes changes in its TWF graph, searches for cycles, and propagates the changes upward, if necessary. A nonleaf controller can receive up-to-date information concerning the TWF graph of its children continuously (i.e., whenever a change occurs) or periodically.

State Recording for Distributed Systems

Quest for A Consistent State Recording Algorithm



1

Naïve State Recording Algorithm

- Assumptions:
  - The system is assumed to be organized in a connected graph topology
  - An edge, often referred in this domain as a channel, connects only two neighboring nodes
  - Channels have infinite capacity

3 May 2024

2

Naïve State Recording Algorithm

1. In one atomic action, the initiator
   1.1 Records its own state;
   1.2 Sends recording messages (RM) to all the neighbors;
2. On receipt of a RM for the first time, every other process,
   2.1 Records its own state;
   2.2 Sends RMs to neighbors except from which it received the RM;

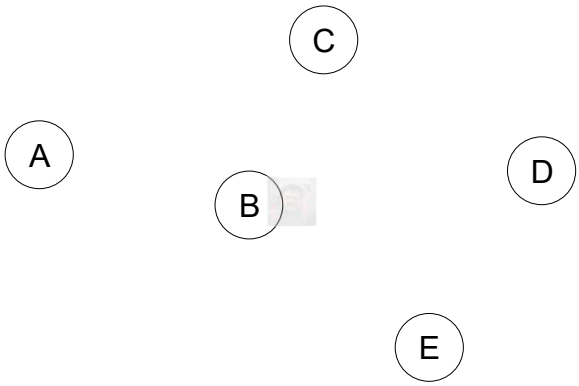
3

Naïve State Recording Algorithm

- The algorithm terminates when
  - Every process, other than the initiator, has received a Recording Message through each incoming channel;
  - No Recording Message is left in any channel.

4

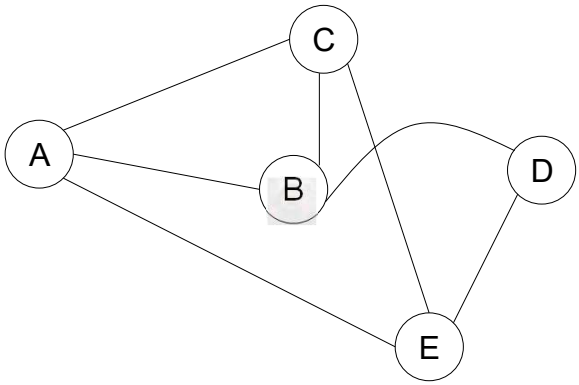
How does it work?



3 May 2024

5

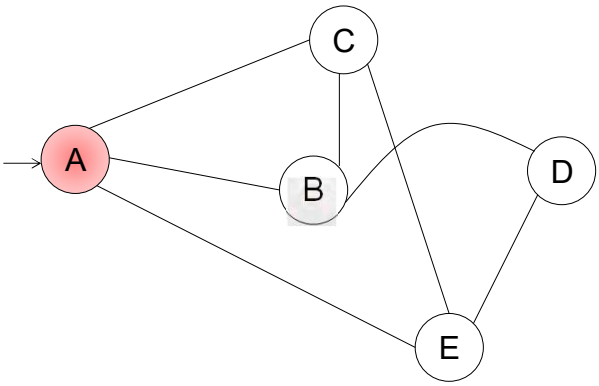
How does it work?



3 May 2024

6

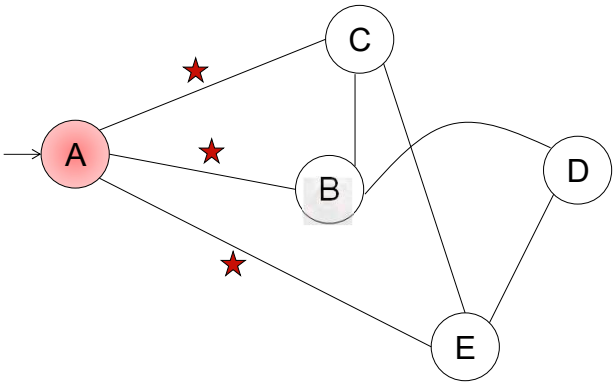
How does it work?



3 May 2024

7

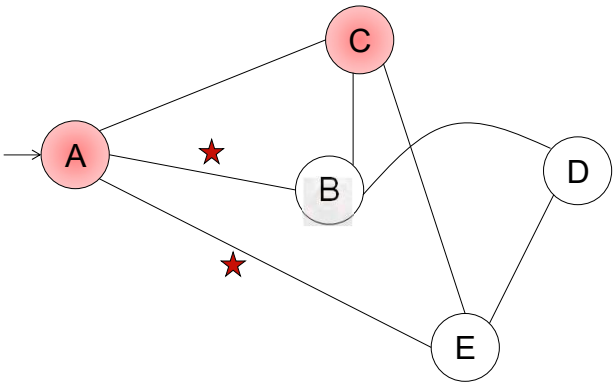
How does it work?



3 May 2024

8

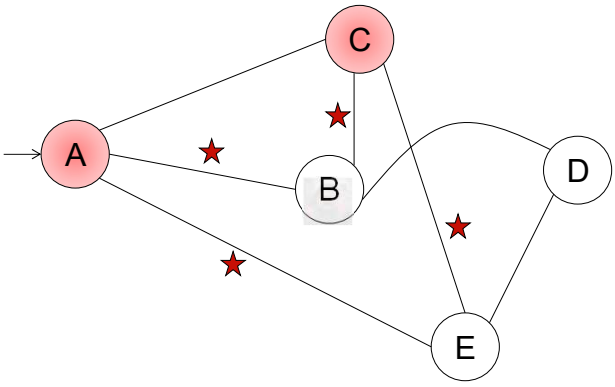
How does it work?



3 May 2024

9

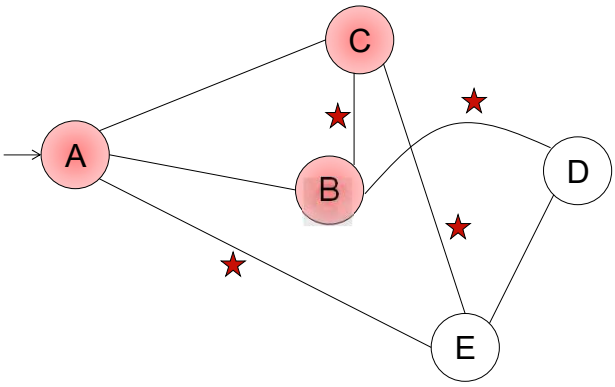
How does it work?



3 May 2024

10

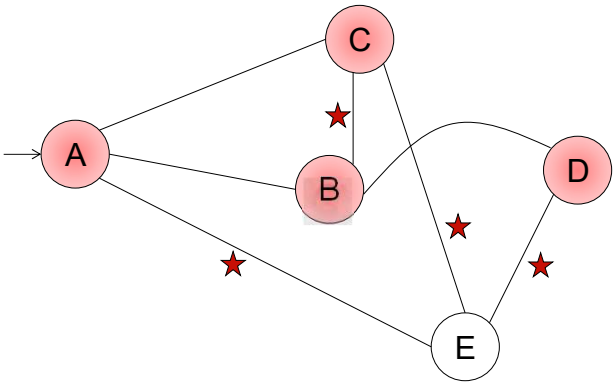
How does it work?



3 May 2024

11

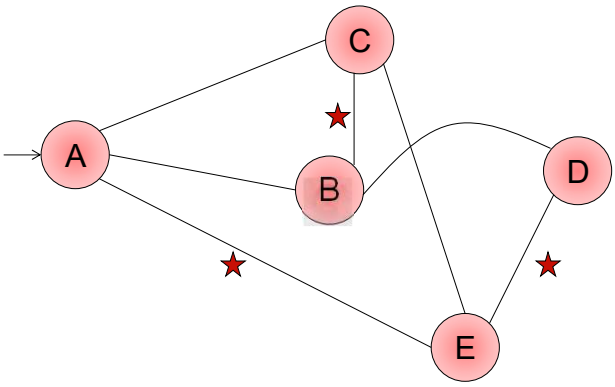
How does it work?



3 May 2024

12

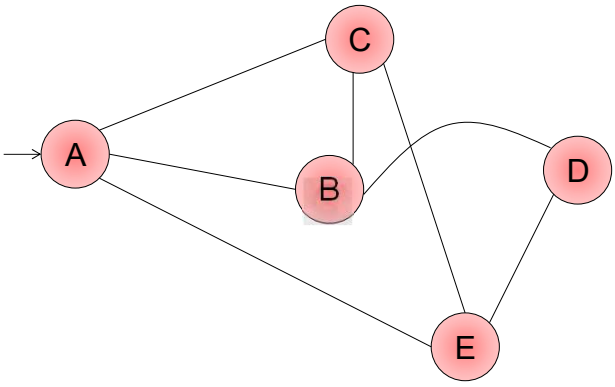
How does it work?



3 May 2024

13

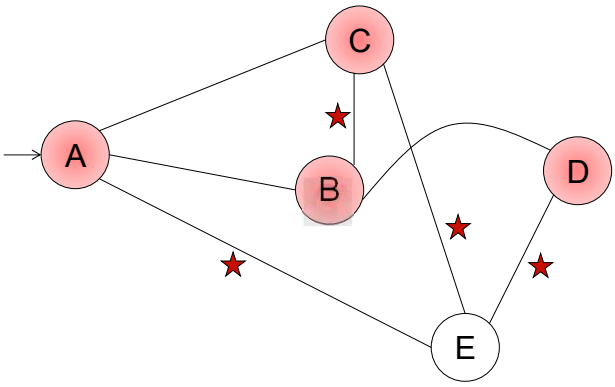
How does it work?



3 May 2024

14

Does it ensure Consistency?

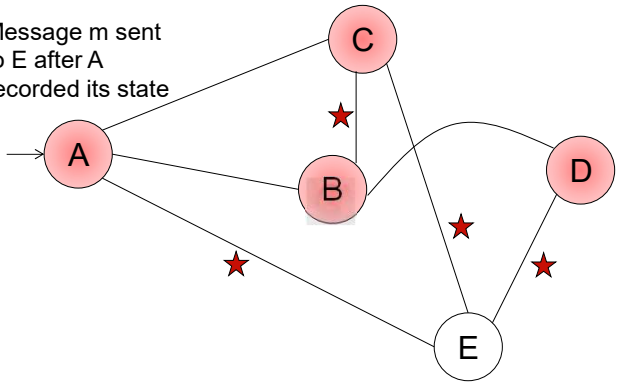


3 May 2024

15

Does it ensure Consistency?

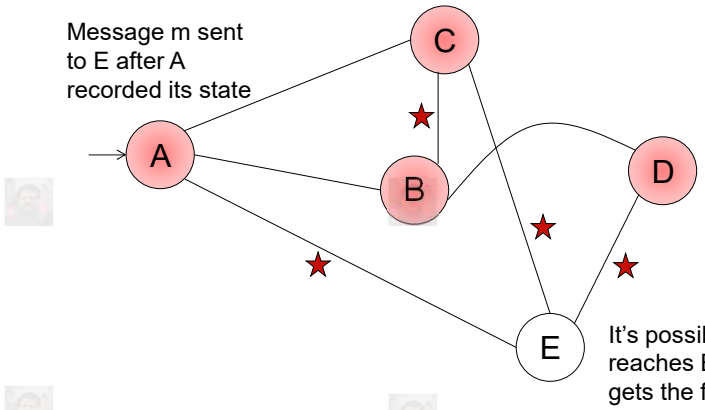
Message m sent
to E after A
recorded its state



3 May 2024

16

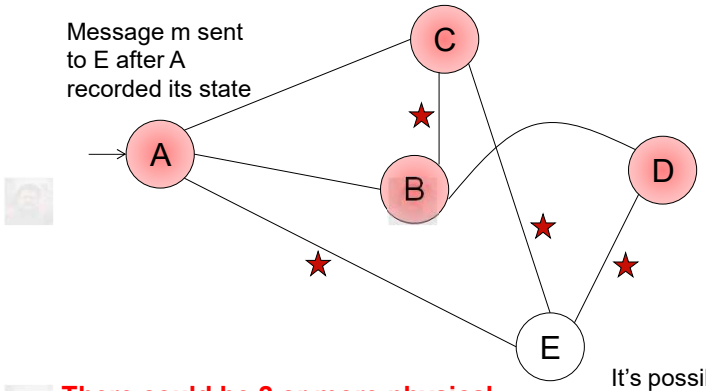
Does it ensure Consistency?



3 May 2024

17

Does it ensure Consistency?



There could be 2 or more physical paths between any pair of nodes

3 May 2024

18

Handling Deadlock for Distributed Systems - I

Different Approaches

1

Outline

- Introduction to Distributed Deadlock Detection
- Control Models for DDD algorithm
- Centralized DDD algorithm
- Diffusion Computation-based DDD algorithm
- Mitchell-Merritt Edge Chasing DDD algorithm

7 June 2024

2

Outline

- Introduction to Distributed Deadlock Detection
- Control Models for DDD algorithm
- Centralized DDD algorithm
- Diffusion Computation-based DDD algorithm
- Mitchell-Merritt Edge Chasing DDD algorithm

Definition

- Deadlock is a state of contention in the system involving two or more blocked processes that can never be resolved unless there is some external intervention.
- Deadlocks are always stable.

Conditions for Deadlock

- Mutual Exclusion: Resource is held by one and only one process at a time
- Hold and Wait: A process is allowed to hold on allocated resources while it's waiting to acquire other resources
- No Preemption
- Circular Wait

7 June 2024

5

Handling Deadlocks

- There are three broad approaches towards mitigating deadlocks either for centralized system or for a distributed system. These are:
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection and Recovery

7 June 2024

6

Deadlock Prevention

- Prioritize processes and assign resources accordingly
- Make prior rules to deny one of the 4 necessary conditions
- May lead to starvation and affect concurrency

7 June 2024

7

Deadlock Avoidance

- Only fulfill those resource requests that won't cause deadlock
- Simulate resource allocation and check if resultant state is safe or not.
- Requires Prior resource requirement information for all processes.
- High cost for scalability

7 June 2024

8

Deadlock Detection and Recovery

- Periodically examine process status and check if one or more processes are in deadlock.
- Select the process to be killed such that it affects least
- Roll back on one or more processes and break the circular wait

9

DD Detection Requirements

- Progress Condition
  - No undetected deadlocks
    - All deadlocks found
    - Deadlocks found in finite time
- Safety Condition
  - No false deadlock detection
    - Phantom deadlocks caused by network latencies
    - Principal problem in building correct DS deadlock detection algorithms

10

Models for Requests

- The AND model requires all resources to be granted to un-block a computation
  - **A cycle is sufficient to declare a deadlock with this model**
- The OR model allows a computation making multiple resource requests to un-block as soon as any one is granted
  - **A cycle is a necessary condition**
  - **A knot is a sufficient condition**

7 June 2024

11

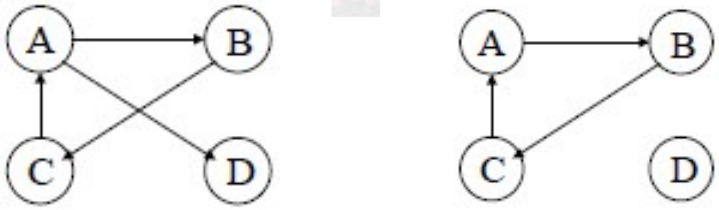
What is a Knot?

- A knot of a graph is a subset K of nodes such that the reachable set of each node in K is exactly K.
- A knot is a cycle with no non-cycle outgoing path from any node.
- In presence of a knot, there will be no active processes to release resources

7 June 2024

12

Cycle and Knot



On left A, B and C are in a Cycle, but not in a Knot

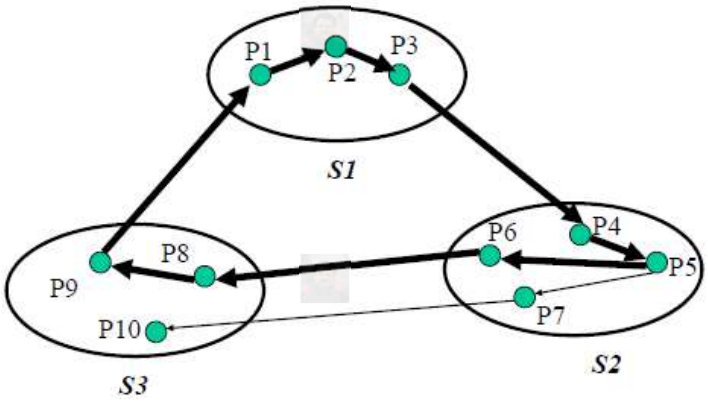
On right A, B and C are in a Knot

In presence of a knot, there will be no active processes to release resources

7 June 2024

13

Cycle and Knot

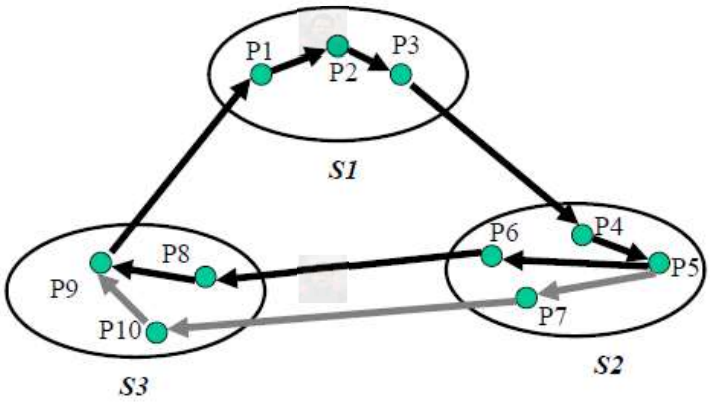


Not a deadlock in the OR model – deadlock in the AND model

7 June 2024

14

Cycle and Knot



A deadlock for both OR and AND model for a Knot

7 June 2024

15

Outline

- Introduction to Distributed Deadlock Detection
- Control Models for DDD algorithm
- Centralized DDD algorithm
- Diffusion Computation-based DDD algorithm
- Mitchell-Merritt Edge Chasing DDD algorithm

7 June 2024

16

Control Models for DDD

- Centralized Control
  - A single control site constructs wait-for graphs (WFGs) and checks for directed cycles.
  - WFG can be maintained continuously (or) built on-demand by requesting WFGs from individual sites.

7 June 2024

17

Control Models for DDD

- Distributed Control
  - WFG is spread over different sites. Any site can initiate the deadlock detection process.
- Hierarchical Control
  - Sites are arranged in a hierarchy.
  - A site checks for cycles only in descendents.

7 June 2024

18

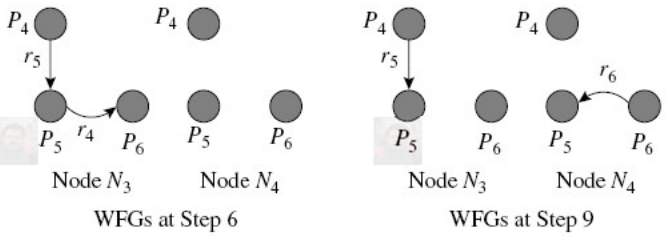
Outline

- Introduction to Distributed Deadlock Detection
- Control Models for DDD algorithm
- Centralized DDD algorithm
- Diffusion Computation-based DDD algorithm
- Mitchell-Merritt Edge Chasing DDD algorithm

Ho-Ramamurthy DDD Algorithm

- Each site maintains 2 status tables: resource status table and process status table.
  - Resource status table: Resources locked by or requested by processes.
  - Process status table: Processes that are locked or are waiting for resources.
- Controller periodically collects these tables from each site.

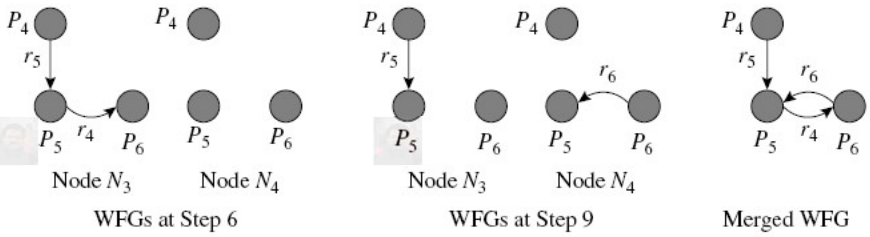
Problem with Centralized Approach



Ho-Ramamurthy DDD Algorithm

- Controller constructs a WFG from transactions common to both the tables.
- If there is no cycle, then no deadlock is detected.
- A cycle means a deadlock.

Problem with Centralized Approach



7 June 2024

23

Ho-Ramamurthy DDD Algorithm

- Single point of failure
- Network congestion issues
- False deadlock detection

7 June 2024

24

Ho-Ramamurthy 2-phase algorithm

- Each site maintains a status table of all processes initiated at that site.
- It includes all resources locked and all resources being waited on.
- Controller site requests (periodically) the status table from each site.
- Controller then constructs WFG from these tables and looks for cycle(s).

25

Ho-Ramamurthy 2-phase algorithm

- If no cycle exists then there is no deadlock.
- If cycle exists, then request for status tables again.
- Construct WFG based only on common transactions in the 2 tables.
- If the same cycle is detected again, system is in deadlock.

26

Ho-Ramamurthy 2-phase algorithm

- Is this approach free from phantom deadlock?

Outline

- Introduction to Distributed Deadlock Detection
- Control Models for DDD algorithm
- Centralized DDD algorithm
- Diffusion Computation-based DDD algorithm
- Mitchell-Merritt Edge Chasing DDD algorithm

Chandy, Misra, Haas Algorithm

- Initiation: Any process, say A, blocked for a long time may initiate the diffusion process as:
  - Send query to all outgoing edges on the WFG
  - Wait for that many replies

Chandy, Misra, Haas Algorithm

- On receipt of an engaging query, a blocked process B does the following:
  - Send query to all outgoing edges on the WFG
  - Wait for that many replies
  - If all the replies arrive and B is blocked continuously since it received the engaging query, then B sends a reply to its parent

Chandy, Misra, Haas Algorithm

- On receipt of a non-engaging query, a blocked process B does the following:
  - If process B is blocked continuously since it received the engaging query, then B sends a dummy non-engaging reply to its parent

31

Chandy, Misra, Haas Algorithm

- Deadlock Detection: If the initiator receives all the replies and it is blocked continuously since it initiated the diffusion process, then a deadlock is detected

32

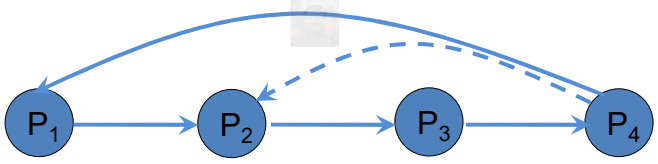
Diffusion-Computation approach

- P_2, P_3 are blocked.
- Hence, P_1 is blocked and sends a query but does not receive a reply because P_4 is not blocked

7 June 2024

33

Diffusion-Computation approach



- Now, P_4 requests for a resource held by P_1 or by P_2.
- In this case, the reply would reach the initiator and a deadlock will be detected.

7 June 2024

34

Thanks for your kind attention

Questions??

35

Handling Deadlock for Distributed Systems - II

Different Approaches

1

Outline

- Introduction to Distributed Deadlock Detection
- Global DDD algorithm
- Centralized DDD algorithm
- Diffusion Computation-based DDD algorithm
- Mitchell-Merritt Edge Chasing DDD algorithm

7 June 2024

2

Mitchell – Merritt DDD Algorithm

- It is an *edge chasing* algorithm where Control messages are sent over WFG edges to detect cycles
- Each process is represented as u/v where u and v are the public and private labels, respectively.
- Labels are initially identical and unique for each node

7 June 2024

3

Mitchell – Merritt DDD Algorithm

- Labels of a process change when it gets blocked on a resource request
- Labels also change when it waits for a process having a larger public label
- A wait-for edge with a specific relation between public and private labels of its source and destination processes indicates presence of a deadlock

7 June 2024

4

State Transitions

- Initiate: Set same random value for u, v. Value for each node need to be unique.
- Block: This will be in effect every time a process is blocked.
  - Add a new block edge in the WFG
  - Set both u, v of the blocked process with k:
    - k = f(u1, u2) yields a unique label greater than both u1 and u2 – the two public labels for the blocking and blocked processes

7 June 2024

5

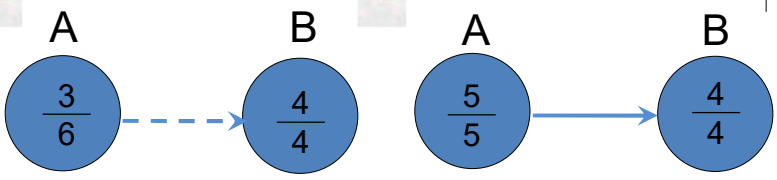
State Transitions

- Transmit: If public label of a blocking process is greater than that of the blocked process in the WFG, then this higher public label propagates in the opposite direction of the edges.
- Detect: If the public and private labels of a blocked process are same, and the value is again same as the public label of the blocking process, a deadlock is detected

7 June 2024

6

Mitchell – Merritt DDD Algorithm

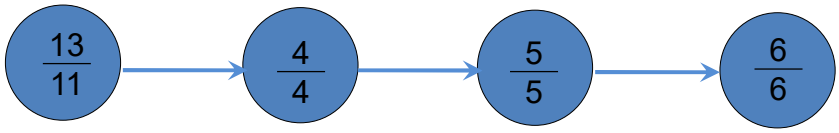


- Block rule
- Assume, f(u1, u2)=Maximum(u1, u2)+1

7 June 2024

7

Mitchell – Merritt DDD Algorithm

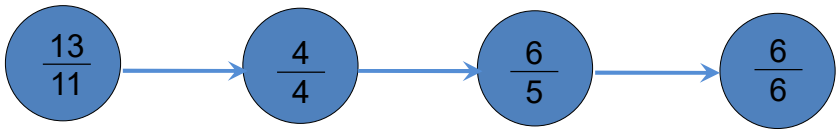


- Transmit rule

7 June 2024

8

Mitchell – Merritt DDD Algorithm

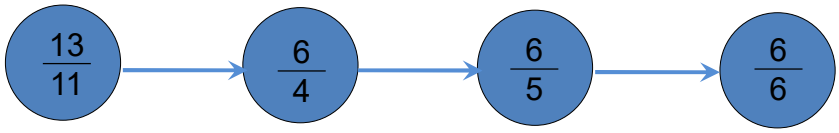


- Transmit rule

7 June 2024

9

Mitchell – Merritt DDD Algorithm

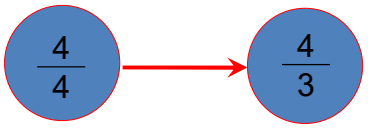


- Transmit rule

7 June 2024

10

Mitchell – Merritt DDD Algorithm

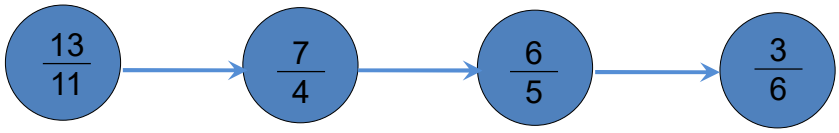


- Detection rule

7 June 2024

11

Mitchell – Merritt DDD Algorithm

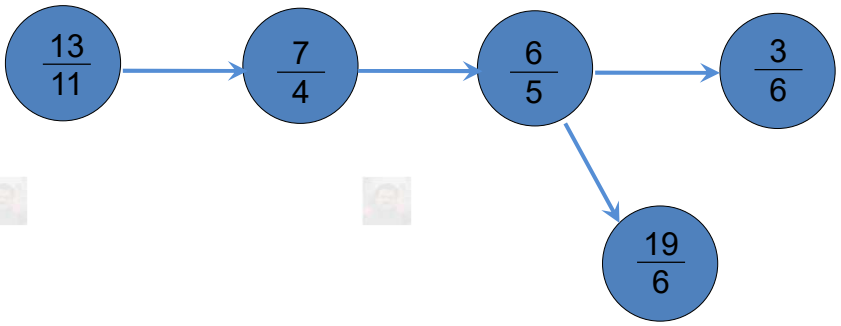


- Scenario 1

7 June 2024

12

Mitchell – Merritt DDD Algorithm

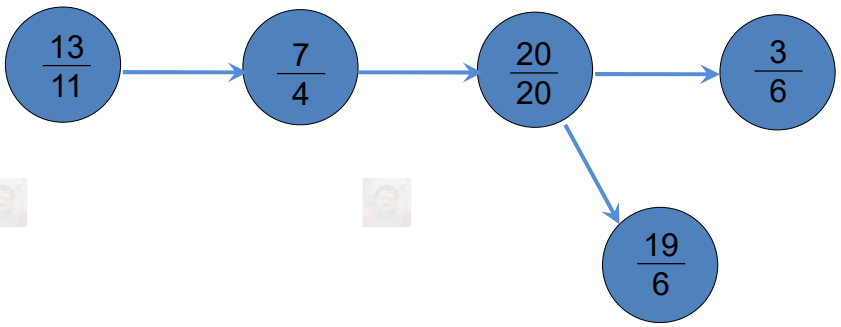


- Scenario 1

7 June 2024

13

Mitchell – Merritt DDD Algorithm

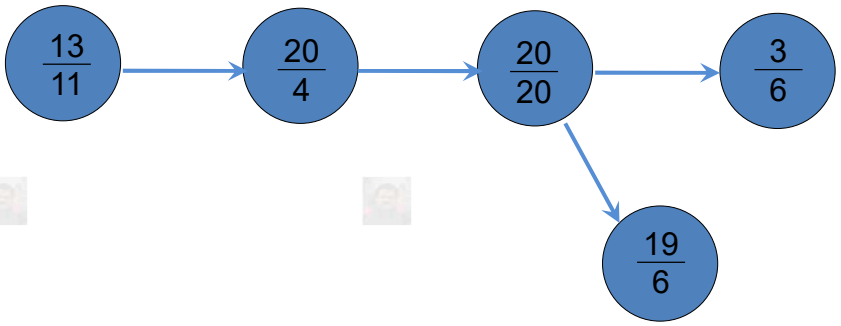


- Scenario 1

7 June 2024

14

Mitchell – Merritt DDD Algorithm

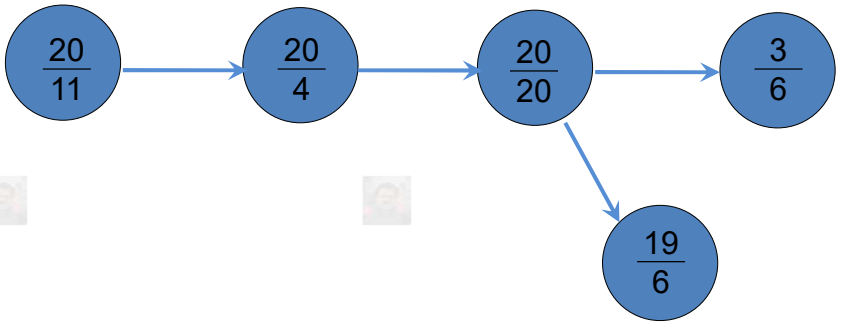


- Scenario 1

7 June 2024

15

Mitchell – Merritt DDD Algorithm

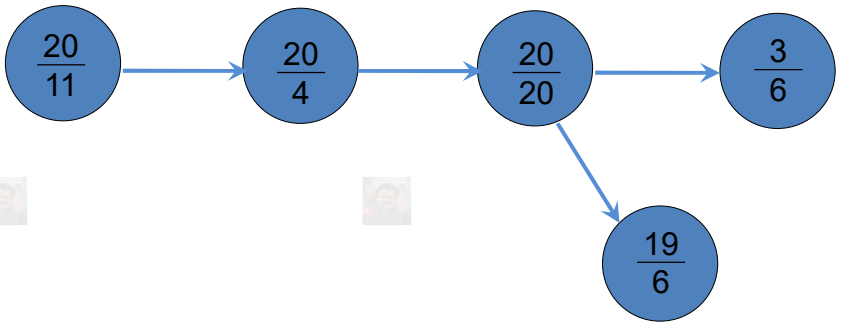


- Scenario 1

7 June 2024

16

Mitchell – Merritt DDD Algorithm

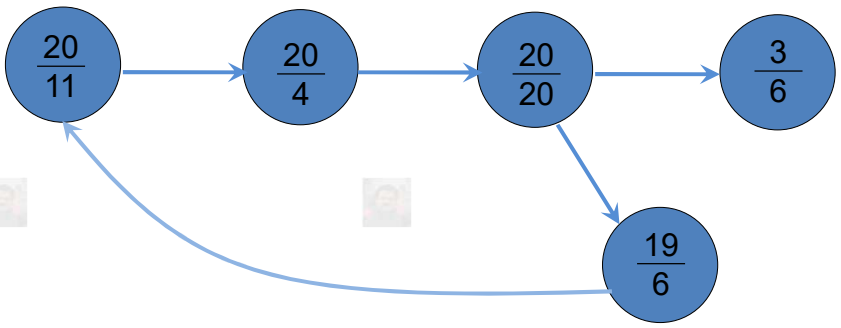


- Scenario 2

7 June 2024

17

Mitchell – Merritt DDD Algorithm

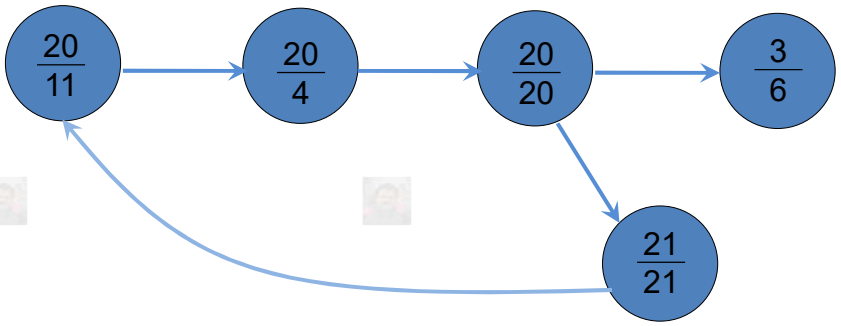


- Scenario 2

7 June 2024

18

Mitchell – Merritt DDD Algorithm

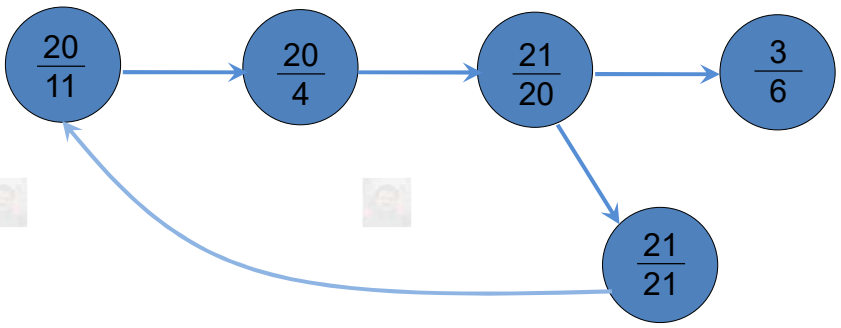


- Scenario 2

7 June 2024

19

Mitchell – Merritt DDD Algorithm

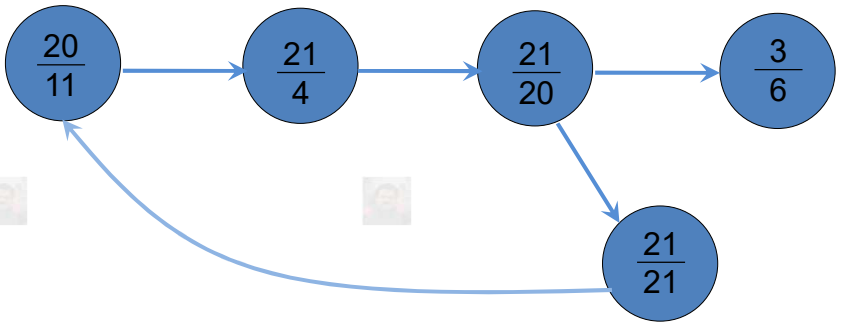


- Scenario 2

7 June 2024

20

Mitchell – Merritt DDD Algorithm

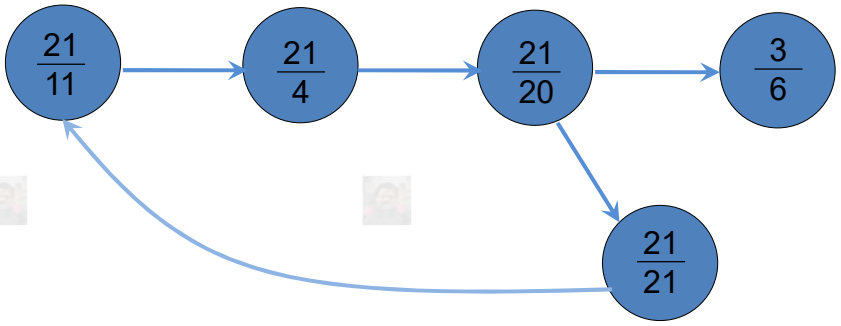


- Scenario 2

7 June 2024

21

Mitchell – Merritt DDD Algorithm

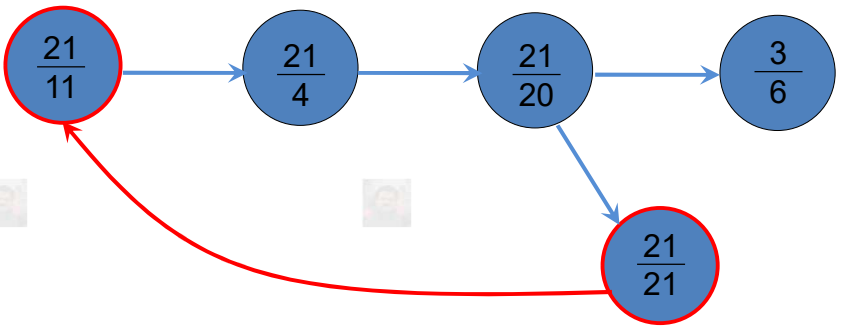


- Scenario 2

7 June 2024

22

Mitchell – Merritt DDD Algorithm

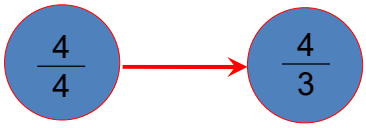


- Scenario 2

7 June 2024

23

Mitchell – Merritt DDD Algorithm



- Detection rule

7 June 2024

24

Few more thoughts….

25

DD Prevention Algorithm

- The basic idea is to ensure that Circular wait does not occur
- Time-stamp creation of a process
  - When process P_k requests a resource allocated to P_m, time-stamps of P_k and P_m are used to decide whether P_k can wait for P_m

7 June 2024

26

DD Prevention Algorithm

- Two approaches
  - Wait-or-die
    - P_i is allowed to wait if older than P_j; otherwise, it is killed
  - Wound-or-wait
    - P_i is allowed to wait if younger than P_j; otherwise P_j is killed
- A killed process retains original timestamp if restarted

7 June 2024

27

Path-pushing vs. Edge Chasing

- Path-pushing:
  - Path information is sent to blocking node
    - e.g., partial WFGs sent to blocking nodes for deadlock detection
    - Obermarck's algorithm
- Edge-chasing:
  - Probe messages sent without path information
    - e.g. Mitchell-Merritt Algorithm

7 June 2024

28

Thanks for your kind attention

Questions??

29

3.4 CODE MIGRATION

So far, we have been mainly concerned with distributed systems in which communication is limited to passing data. However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system. In this section, we take a detailed look at what code migration actually is. We start by considering different approaches to code migration, followed by a discussion on how to deal with the local resources that a migrating program uses. A particularly hard problem is migrating code in heterogeneous systems, which is also discussed. To make matters concrete, we discuss the D'Agents system for mobile agents at the end of this section. Note that security issues concerning code migration are deferred to Chap. 8.

3.4.1 Approaches to Code Migration

Before taking a look at the different forms of code migration, let us first consider why it may be useful to migrate code.

Reasons for Migrating Code

Traditionally, code migration in distributed systems took place in the form of **process migration** in which an entire process was moved from one machine to another. Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so. That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

Load distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of processors, play an important role in compute-intensive systems. However, in many modern distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication. Moreover, due to the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration is often based on qualitative reasoning instead of mathematical models.

Consider, for example, a client-server system in which the server manages a huge database. If a client application needs to do many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

This same reason can be used for migrating parts of the server to the client. For example, in many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations. Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network. The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.

Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance.

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and deciding in advance where each part should be executed. This approach, for example, has led to the different multitiered client-server applications discussed in Chap. 1.

However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application. This approach requires that the software be readily available to the client at the time the client application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. 3-7. This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

The important advantage of this model of dynamically downloading client-side software, is that clients need not have all the software preinstalled to talk to servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on

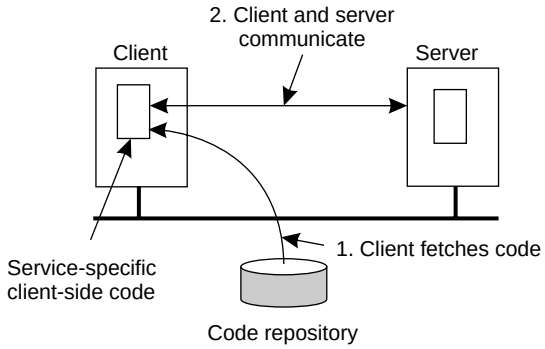


Figure 3-7. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

the server. There are, of course, also disadvantages. The most serious one, which we discuss in Chap. 8, has to do with security. Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea.

Models for Code Migration

Although code migration suggests that we move only code between machines, the term actually covers a much richer area. Traditionally, communication in distributed systems is concerned with exchanging data between processes. Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target. In some cases, as in process migration, the execution status of a program, pending signals, and other parts of the environment must be moved as well.

To get a better understanding of the different models for code migration, we use a framework described in (Fugetta et al., 1998). In this framework, a process consists of three segments. The *code segment* is the part that contains the set of instructions that make up the program that is being executed. The *resource segment* contains references to external resources needed by the process, such as files, printers, devices, other processes, and so on. Finally, an *execution segment* is used to store the current execution state of a process, consisting of private data, the stack, and the program counter.

The bare minimum for code migration is to provide only **weak mobility**. In this model, it is possible to transfer only the code segment, along with perhaps some initialization data. A characteristic feature of weak mobility is that a transferred program is always started from its initial state. This is what happens, for example, with Java applets. The benefit of this approach is its simplicity.

Weak mobility requires only that the target machine can execute that code, which essentially boils down to making the code portable. We return to these matters when discussing migration in heterogeneous systems.

In contrast to weak mobility, in systems that support **strong mobility** the execution segment can be transferred as well. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off. Clearly, strong mobility is much more powerful than weak mobility, but also much harder to implement. An example of a system that supports strong mobility is D'Agents, which we discuss later in this section.

Irrespective of whether mobility is weak or strong, a further distinction can be made between sender-initiated and receiver-initiated migration. In **sender-initiated** migration, migration is initiated at the machine where the code currently resides or is being executed. Typically, sender-initiated migration is done when uploading programs to a compute server. Another example is sending a search program across the Internet to a Web database server to perform the queries at that server. In **receiver-initiated** migration, the initiative for code migration is taken by the target machine. Java applets are an example of this approach.

Receiver-initiated migration is often simpler to implement than sender-initiated migration. In many cases, code migration occurs between a client and a server, where the client takes the initiative for migration. Securely uploading code to a server, as is done in sender-initiated migration, often requires that the client has previously been registered and authenticated at that server. In other words, the server is required to know all its clients, the reason being is that the client will presumably want access to the server's resources such as its disk. Protecting such resources is essential. In contrast, downloading code as in the receiver-initiated case, can often be done anonymously. Moreover, the server is generally not interested in the client's resources. Instead, code migration to the client is done only for improving client-side performance. To that end, only a limited number of resources need to be protected, such as memory and network connections. We return to secure code migration extensively in Chap. 8.

In the case of weak mobility, it also makes a difference if the migrated code is executed by the target process, or whether a separate process is started. For example, Java applets are simply downloaded by a Web browser and are executed in the browser's address space. The benefit of this approach is that there is no need to start a separate process, thereby avoiding communication at the target machine. The main drawback is that the target process needs to be protected against malicious or inadvertent code executions. A simple solution is to let the operating system take care of that by creating a separate process to execute the migrated code. Note that this solution does not solve the resource-access problems just mentioned.

Instead of moving a running process, also referred to as process migration, strong mobility can also be supported by remote cloning. In contrast to process

migration, cloning yields an exact copy of the original process, but now running on a different machine. The cloned process is executed in parallel to the original process. In UNIX systems, remote cloning takes place by forking off a child process and letting that child continue on a remote machine. The benefit of cloning is that the model closely resembles the one that is already used in many applications. The only difference is that the cloned process is executed on a different machine. In this sense, migration by cloning is a simple way to improve distribution transparency.

The various alternatives for code migration are summarized in Fig. 3-8.

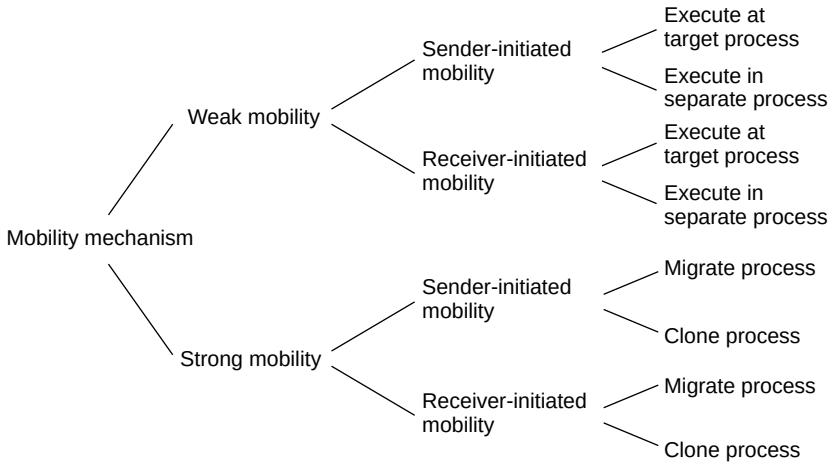


Figure 3-8. Alternatives for code migration.

3.4.2 Migration and Local Resources

So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult, is that the resource segment cannot always be simply transferred along with the other segments without being changed. For example, suppose a process holds a reference to a specific TCP port through which it was communicating with other (remote) processes. Such a reference is held in its resource segment. When the process moves to another location, it will have to give up the port and request a new one at the destination. In other cases, transferring a reference need not be a problem. For example, a reference to a file by means of an absolute URL will remain valid irrespective of the machine where the process that holds the URL resides.

To understand the implications that code migration has on the resource segment, Fuggetta et al. distinguish three types of process-to-resource bindings. The strongest binding is when a process refers to a resource by its identifier. In that

case, the process requires precisely the referenced resource, and nothing else. An example of such a **binding by identifier** is when a process uses a URL to refer to a specific Web site or when it refers to an FTP server by means of that server's Internet address. In the same line of reasoning, references to local communication endpoints also lead to a binding by identifier.

A weaker form of process-to-resource binding is when only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide that same value. A typical example of **binding by value** is when a program relies on standard libraries, such as those for programming in C or Java. Such libraries should always be locally available, but their exact location in the local file system may differ between sites. Not the specific files, but their content is important for the proper execution of the process.

Finally, the weakest form of binding is when a process indicates it needs only a resource of a specific type. This **binding by type** is exemplified by references to local devices, such as monitors, printers, and so on.

When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding. If, and exactly how a reference should be changed, depends on whether that resource can be moved along with the code to the target machine. More specifically, we need to consider the resource-to-machine bindings, and distinguish the following cases. **Unattached resources** can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated. In contrast, moving or copying a **fastened resource** may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites. Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment. Finally, **fixed resources** are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices. Another example of a fixed resource is a local communication endpoint.

Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code. These nine combinations are shown in Fig. 3-9.

Let us first consider the possibilities when a process is bound to a resource by identifier. When the resource is unattached, it is generally best to move it along with the migrating code. However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries. An example of such a reference is a URL. When the resource is fastened or fixed, the best solution is also to establish a global reference.

It is important to realize that establishing a global reference may be more than just making use of URLs, and that the use of such a reference is sometimes prohibitively expensive. Consider, for example, a program that generates high-quality

Process-to-resource binding	Resource-to-machine binding: Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV,GR)	GR (or CP)	GR
By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR Establish a global systemwide reference
MV Move the resource
CP Copy the value of the resource
RB Rebind process to locally available resource

Figure 3-9. Actions to be taken with respect to the references to local resources when migrating code to another machine.

images for a dedicated multimedia workstation. Fabricating high-quality images in real time is a compute-intensive task, for which reason the program may be moved to a high-performance compute server. Establishing a global reference to the multimedia workstation means setting up a communication path between the compute server and the workstation. In addition, there is significant processing involved at both the server and the workstation to meet the bandwidth requirements of transferring the images. The net result may be that moving the program to the compute server is not such a good idea, only because the cost of the global reference is too high.

Another example of where establishing a global reference is not always that easy, is when migrating a process that is making use of a local communication endpoint. In that case, we are dealing with a fixed resource to which the process is bound by the identifier. There are basically two solutions. One solution is to let the process set up a connection to the source machine after it has migrated and install a separate process at the source machine that simply forwards all incoming messages. The main drawback of this approach is that whenever the source machine malfunctions, communication with the migrated process may fail. The alternative solution is to have all processes that communicated with the migrating process, change *their* global reference, and send messages to the new communication endpoint at the target machine.

The situation is different when dealing with bindings by value. Consider first a fixed resource. The combination of a fixed resource and binding by value occurs, for example, when a process assumes that memory can be shared between processes. Establishing a global reference in this case would mean that we need to implement distributed shared memory mechanisms as discussed in Chap. 1. Obviously, this is not really a viable solution.

Fastened resources that are referred to by their value, are typically runtime libraries. Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place. Establishing a global reference is a better alternative when huge amounts of data are to

be copied, as may be the case with dictionaries and thesauruses in text processing systems.

The easiest case is when dealing with unattached resources. The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes. In the latter case, establishing a global reference is the only option.

The last case deals with bindings by type. Irrespective of the resource-to-machine binding, the obvious solution is to rebind the process to a locally available resource of the same type. Only when such a resource is not available, will we need to copy or move the original one to the new destination, or establish a global reference.

3.4.3 Migration in Heterogeneous Systems

So far, we have tacitly assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous systems. In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform, perhaps after recompiling the original source. Also, we need to ensure that the execution segment can be properly represented at each platform.

Problems can be somewhat alleviated when dealing only with weak mobility. In that case, there is basically no runtime information that needs to be transferred between machines, so that it suffices to compile the source code, but generate different code segments, one for each potential target platform.

In the case of strong mobility, the major problem that needs to be solved is the transfer of the execution segment. The problem is that this segment is highly dependent on the platform on which the process is being executed. In fact, only when the target machine has the same architecture and is running exactly the same operating system, is it possible to migrate the execution segment without having to alter it.

The execution segment contains data that is private to the process, its current stack, and the program counter. The stack will partly consist of temporary data, such as values of local variables, but may also contain platform-dependent information such as register values. The important observation is that if we can avoid having execution depend on platform-specific data, it would be much easier to transfer the segment to a different machine, and resume execution there.

A solution that works for procedural languages such as C and Java is shown in Fig. 3-10 and works as follows. Code migration is restricted to specific points in the execution of a program. In particular, migration can take place only when a next subroutine is called. A subroutine is a function in C, a method in Java, and so

on. The runtime system maintains its own copy of the program stack, but in a machine-independent way. We refer to this copy as the **migration stack**. The migration stack is updated when a subroutine is called, or when execution returns from a subroutine.

When a subroutine is called, the runtime system marshals the data that have been pushed onto the stack since the last call. These data represent values of local variables, along with parameter values for the newly called subroutine. The marshaled data are then pushed onto the migration stack, along with an identifier for the called subroutine. In addition, the address where execution should continue when the caller returns from the subroutine is pushed in the form of a jump label onto the migration stack as well.

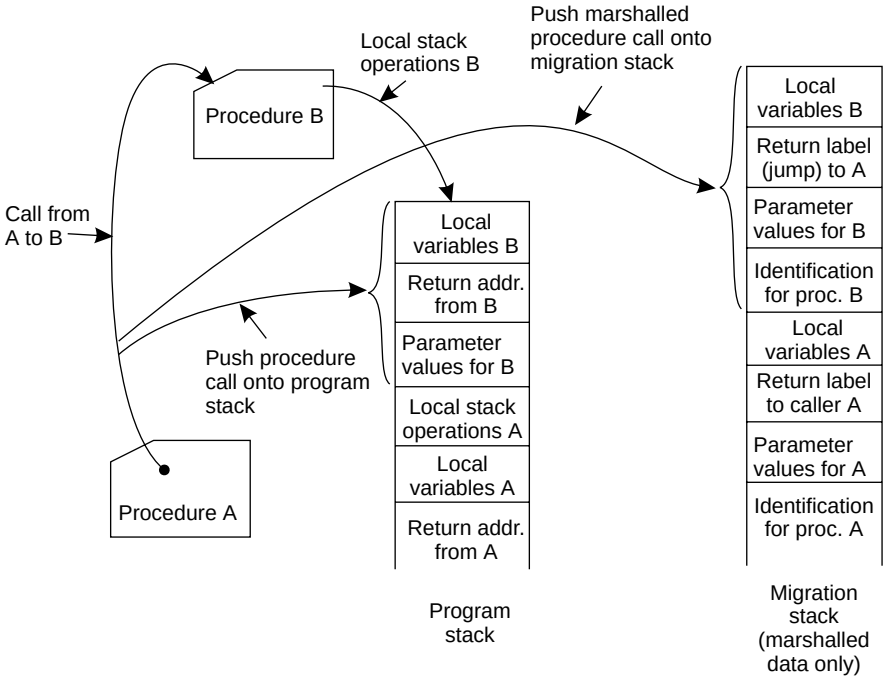


Figure 3-10. The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment.

If code migration takes place at the point where a subroutine is called, the runtime system first marshals all global program-specific data forming part of the execution segment. Machine-specific data are ignored, as well as the current stack. The marshaled data are transferred to the destination, along with the migration stack. In addition, the destination loads the appropriate code segment containing the binaries fit for its machine architecture and operating system. The marshaled data belonging to the execution segment are unmarshaled, and a new

runtime stack is constructed by unmarshaling the migration stack. Execution can then be resumed by simply entering the subroutine that was called at the original site.

It is clear that this approach works only if the compiler generates code to update the migration stack whenever a subroutine is entered or exited. The compiler also generates labels in the caller's code allowing a return from a subroutine to be implemented as a (machine-independent) jump. In addition, we also need a suitable runtime system. Nevertheless, there are a number of systems that have successfully exploited these techniques. For example, Dimitrov and Rego (1998) show how migration of C/C++ programs in heterogeneous systems can be supported by slightly modifying the language, and using only a preprocessor to insert the necessary code to maintain the migration stack.

The problems coming from heterogeneity are in many respects the same as those of portability. Not surprisingly, solutions are also very similar. For example, at the end of the 1970s, a simple solution to alleviate many of the problems of porting Pascal to different machines was to generate machine-independent intermediate code for an abstract virtual machine (Barron, 1981). That machine, of course, would need to be implemented on many platforms, but it would then allow Pascal programs to be run anywhere. Although this simple idea was widely used for some years, it never really caught on as the general solution to portability problems for other languages, notably C.

About 20 years later, code migration in heterogeneous systems is being attacked by scripting languages and highly portable languages such as Java. All such solutions have in common that they rely on a virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java). Being in the right place at the right time is also important for language developers.

The only serious drawback of the virtual-machine approach is that we are generally stuck with a specific language, and it is often not one that has been used before. For this reason, it is important that languages for mobility provide interfaces to existing languages.

3.4.4 Example: D'Agents

To illustrate code migration, let us now take a look at a middleware platform that supports various forms of code migration. D'Agents formerly called Agent Tcl, is a system that is built around the concept of an agent. An agent in D'Agents is a program that can migrate between machines in a heterogeneous system. Here, we concentrate only on the migration capabilities of D'Agents, and return to a more general discussion on software agents in the next section. Also, we ignore the security of the system and defer further discussion to Chap. 8. More information on D'Agents can be found in (Gray, 1996b; Kotz et al., 1997).

Overview of Code Migration in D'Agents

An agent in D'Agents is a program that can migrate between different machines. In principle, programs can be written in any language, as long as the target machine can execute the migrated code. In practice, this means that programs in D'Agents are written in an interpretable language, notably, the Tool Command Language, that is, Tcl (Ousterhout, 1994), Java, or Scheme (Rees and Clinger, 1986). Using only interpretable languages makes it much easier to support heterogeneous systems.

A program, or agent, is executed by a process running the interpreter for the language in which the program is written. Mobility is supported in three different ways: sender-initiated weak mobility, strong mobility by process migration, and strong mobility by process cloning.

Weak mobility is implemented by means of the agent_submit command. An identifier of the target machine is given as a parameter, as well as a *script* that is to be executed at that machine. A script is nothing but a sequence of instructions. The script is transferred to the target machine along with any procedure definitions and copies of variables that the target machine needs to execute the script. At the target machine, a process running the appropriate interpreter is subsequently started to execute the script. In terms of the alternatives for code migration mentioned in Fig. 3-8, D'Agents thus provides support for sender-initiated weak mobility, where the migrated code is executed in a separate process.

To give an example of weak mobility in D'Agents, Fig. 3-11 shows part of a simple Tcl agent that submits a script to a remote machine. In the agent, the procedure factorial takes a single parameter and recursively evaluates the expression that calculates the factorial of its parameter value. The variables *number* and *machine* are assumed to be properly initialized (e.g., by asking the user for values), after which the agent calls agent_submit. The script

```
factorial $number
```

is sent to the target machine referred to by the variable *machine*, along with the description of the procedure *factorial* and the initial value of the variable *number*. D'Agents automatically arranges that results are sent back to the agent. The call to agent_receive establishes that the submitting agent is blocked until the results of the calculation have been received.

Sender-initiated strong mobility is also supported, both in the form of process migration and process cloning. To migrate a running agent, the agent calls agent_jump specifying the target machine to which it should migrate. When agent_jump is called, execution of the agent on the source machine is suspended and its resource segment, code segment, and execution segment are marshaled into a message that is subsequently sent to the target machine. Upon arrival of that message, a new process running the appropriate interpreter is started. That process unmarshals the message and continues at the instruction following the previous

```
proc factorial n {
    if { $n ≤ 1 } { return 1; }                    # fac(1) = 1
    expr $n * [ factorial [ expr $n – 1] ]         # fac(n) = n * fac(n–1)
}

set number ...          # tells which factorial to compute
set machine ...         # identify the target machine

agent_submit $machine –procs factorial –vars number –script { factorial $number }

agent_receive ...       # receive the results (left unspecified for simplicity)
```

Figure 3-11. A simple example of a Tcl agent in D'Agents submitting a script to a remote machine (adapted from Gray, 1995).

call to agent_jump. The process that was running the agent at the source machine, exits.

```
proc all_users machines {
    set list ""                    # Create an initially empty list
    foreach m $machines {          # Consider all hosts in the set of given machines
        agent_jump $m              # Jump to each host
        set users [exec who]       # Execute the who command
        append list $users         # Append the results to the list
    }
    return $list                   # Return the complete list when done
}
set machines ...                   # Initialize the set of machines to jump to
set this_machine ...               # Set to the host that starts the agent

# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in $machines.
agent_submit $this_machine  –procs all_users –vars machines \
                            –script { all_users $machines }

agent_receive ...                  # receive the results (left unspecified for simplicity)
```

Figure 3-12. An example of a Tcl agent in D'Agents migrating to different machines where it executes the UNIX *who* command (adapted from Gray, 1995).

An example of agent migration is given in Fig. 3-12, which shows a simplified version of an agent that finds out which users are currently logged in by executing the UNIX command *who* on each host. The behavior of the agent is given

by the procedure all_users. It maintains a list of users that is initially empty. The set of hosts that it should visit is given by the parameter *machines*. The agent jumps to each host, puts the results of executing *who* in the variable *users*, and appends that to its list. In the main program, the agent is created on the current machine by submission, that is, using the previously discussed mechanisms for weak mobility. In this case, agent_submit is requested to execute the script

```
all_users $machines
```

and is given the procedure and set of hosts as additional parameters.

Finally, process cloning is supported by means of the agent_fork command. This command behaves almost the same as agent_jump, except that the process running the agent at the source machine simply continues with the instruction following its call to agent_fork. Like the fork operation in UNIX, agent_fork returns a value by which the caller can check whether it is the cloned version (corresponding to the ''child'' in UNIX), or the original caller (i.e., the ''parent'').

Implementation Issues

To explain some of the internal implementation details, consider agents that have been written in Tcl. Internally, the D'Agents systems consists of five layers, as shown in Fig. 3-13. The lowest layer is comparable to Berkeley sockets in the sense that it implements a common interface to the communication facilities of the underlying network. In D'Agents, it is assumed that the underlying system provides facilities for handling TCP messages and e-mail.

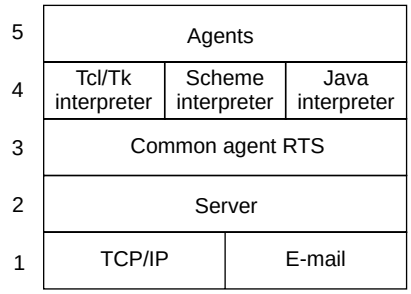


Figure 3-13. The architecture of the D'Agents system.

The next layer consists of a server that runs at each machine where D'Agents agents are executing. The server is responsible for agent management, authentication, and management of communication between agents. For the latter, the server assigns a location-unique identifier to each agent. Using the network address of the server, each agent can then be referred to by an *(address, local-id)*-pair. This low-level name is used to set up communication between two agents.

The third layer is at the heart of the D'Agents system, and consists of a language-independent core that supports the basic model of agents. For example, this layer contains implementations to start and end an agent, implementations of the various migration operations, and facilities for interagent communication. Clearly, the core operates closely with the server, but, in contrast to the server, is not responsible for managing a collection of agents running on the same machine.

The fourth layer consists of interpreters, one for each language supported by D'Agents. Each interpreter consists of a component for language interpretation, a security module, an interface to the core layer, and a separate module to capture the state of a running agent. This last module is essential for supporting strong mobility, and is discussed in more detail below.

The highest-level layer consists of agents written in one of the supported languages. Each agent in D'Agents is executed by a separate process. For example, when an agent migrates to machine *A*, the server there forks a process that will execute the appropriate interpreter for the migrating agent. The new process is then handed the state of the migrating agent, after which it continues where the agent had previously left off. The server keeps track of the processes it created using a local pipe, so that it can pass incoming calls to the appropriate process.

The more difficult part in the implementation of D'Agents, is capturing the state of a running agent and shipping that state to another machine. In the case of Tcl, the state of an agent consists of the parts shown in Fig. 3-14. Essentially, there are four tables containing global definitions of variables and scripts, and two stacks for keeping track of the execution status.

State	Description
Global interpreter variables	Variables needed by the interpreter of an agent
Global system variables	Return codes, error codes, error strings, etc.
Global program variables	User-defined global variables in a program
Procedure definitions	Definitions of scripts to be executed by an agent
Stack of commands	Stack of commands currently being executed
Stack of call frames	Stack of activation records, one for each running command

Figure 3-14. The parts comprising the state of an agent in D'Agents.

There is a table for storing global variables needed by the interpreter. For example, there may be an event handler telling the interpreter which procedure to call when a message from a specific agent arrives. Such an *(event, handler)*-pair is stored in the interpreter table. Another table contains global system variables for storing error codes, error strings, result codes, result strings, etc. There is also a separate table containing all user-defined global program variables. Finally, a separate table contains the definitions of the procedures associated with an agent. These procedure definitions need to migrate along with the agent in order to allow interpretation at the target machine.

The more interesting parts related to agent migration are the two stacks by which an accurate account is kept of the actual execution status of an agent. Basically, an agent is considered as a series of Tcl commands, possibly embedded in constructions such as loops, case statements, and so on. In addition, commands may be grouped into procedures. As is normal for any interpreted language, an agent is executed command by command.

First consider what happens when a basic Tcl command is executed, that is, a command that is not a call to a user-defined procedure. The interpreter parses the command and builds a record that is to be pushed onto what is called the **command stack**. Such a record contains all the necessary fields to actually execute the command, such as its parameter values, a pointer to a procedure implementing the command, and so on. This record is then pushed onto the stack, after which it can be handed over to the component responsible for actually executing the command. In other words, the command stack gives a precise account of the current execution status of an agent.

Tcl also supports user-defined procedures. In addition to the command stack, the runtime environment of D'Agents keeps track of a stack of activation records, also called call frames. A call frame in D'Agents contains a table of variables local to the procedure, along with the names and values of the parameters by which the procedure was called. A call frame is created only as the result of a procedure call, and as such is related to a procedure-call command as pushed onto the command stack. The call frame keeps a reference to its associated command.

Now consider what happens, for example, when an agent calls agent_jump, by which the agent migrates to another machine. At that point, the complete state of the agent as just described is marshaled into a series of bytes. In other words, all four tables and the two stacks are put together into a single array of bytes and shipped to the target machine. The D'Agents server on the target machine subsequently creates a new process running the Tcl interpreter. That process is handed the marshaled data, which it then unmarshals into the state the agent was in when it called agent_jump. By simply popping the command from the top of the command stack, execution continues exactly where it had left off.

Distributed File System

Case study with HDFS

1

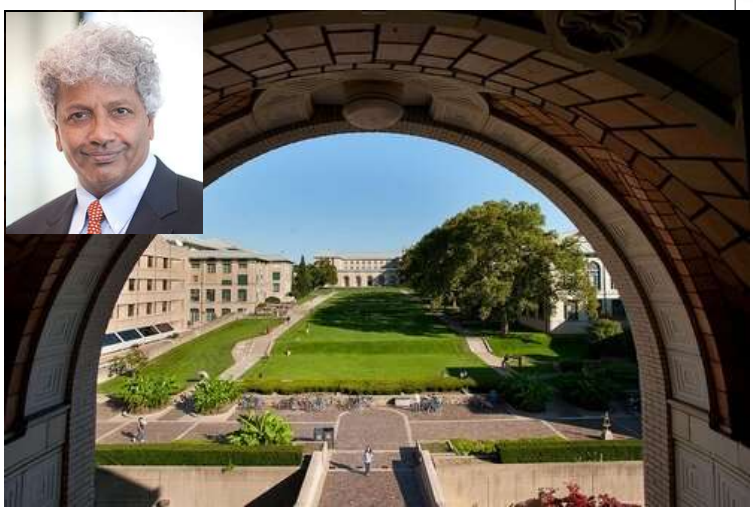
Outline

- Cross-Domain Communication
- System-wide DFS
- Client-Server Architecture
- Case Study: HDFS

16 July 2024

2

Problem of too many…



3

Outline

- Cross-Domain Communication
- System-wide DFS
- Client-Server Architecture
- Case Study: HDFS

4

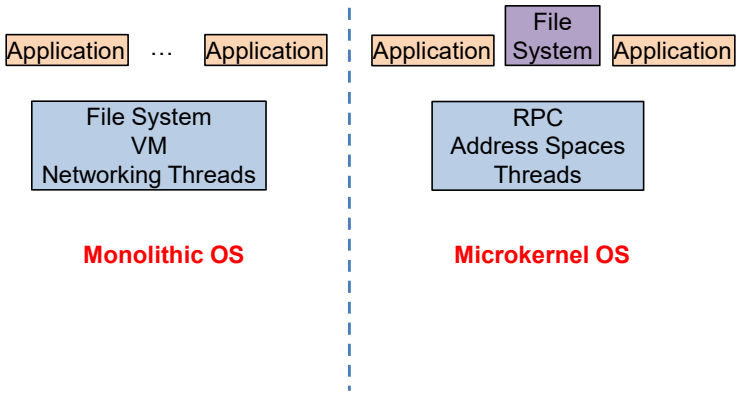
Cross-Domain Communication

- Options for cross-domain communication:
  - Shared memory with Semaphores, monitors, etc.,…
  - Remote Procedure Call
  - System-wide File System
- Examples of RPC based systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

16 July 2024

5

OS Architecture

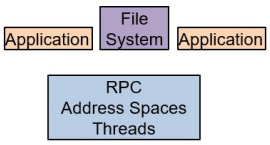


16 July 2024

6

Microkernel Architecture and DFS

- In Microkernel architecture, the File system looks remote.



- Distributed File System (DFS) aims to support transparent access to files on remote disks.
- A single, global name-space, if possible, will support every file with a unique name.
- Location Transparency: Services may migrate, and files can move without involving the user.

16 July 2024

7

Outline

- Cross-Domain Communication
- System-wide DFS
- Client-Server Architecture
- Case Study: HDFS

16 July 2024

8

Architecture Options

- Fully distributed with files in all sites:
  - Issues
    - Performance
    - Implementation complexity
- Client-server Model:
  - File server
    - Dedicated sites storing files perform storage and retrieval operations
  - Client
    - Rest of the sites access the files

16 July 2024

9

Mounting of Remote FS

- System manager mounts remote file system by giving name and local mount point
- Transparency to user - all reads and writes look like local reads and writes to user
  - e.g., /users/sue/foo→lmp/sue/foo on server
- Only previously mounted remote file systems can be accessed transparently

16 July 2024

10

Mounting of Remote FS

- The mount mechanism binds together several filename spaces into a single hierarchically structured name space
- A name space ‘A’ can be mounted (bounded) at an internal node (mount point) of a name space ‘B’
- Kernel requires to maintain the *mount table,* a mapping between mount points to storage devices

16 July 2024

11

Mount Information at Client Side

- A client mounts other file systems
- Different clients may not see the same filename space
- If a file moves to another server, every client needs to update its mount table

16 July 2024

12

Mount Information at Server Side

- Server mounts the file systems of remote nodes
- Every client sees the same filename space
- If a file moves to another server, mounting information at the server only needs to change

Andrew File System (AFS)

- Andrew file system (AFS) is a location-independent file system that uses a local cache to reduce the workload and increase the performance of a distributed computing environment.
  - Introduced by researchers at Carnegie-Mellon University (CMU) in 1983. Team led by Prof. M. Satyanarayanan of CMU.

Why AFS?

- Initial goal of AFS was wide accessibility of computational facilities
- An integrated, campus-wide file system with functional characteristics as close to that of UNIX as possible.
- To enable a student to sit down at any workstation and start using his or her files with as little hassle as possible.

16 July 2024

15

Why AFS?

- They did not want to modify existing application programs, which assume a UNIX file system, in any way.
- Thus, the first design choice was to make the file system compatible with UNIX at the system call level.

16 July 2024

16

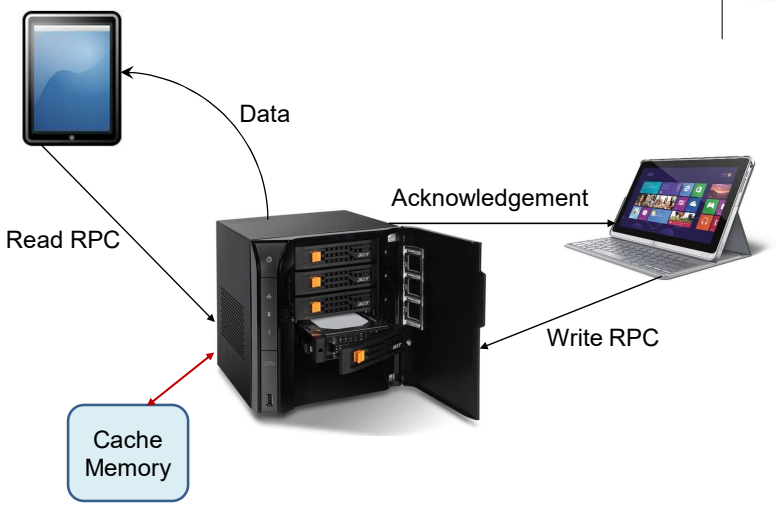
Outline

- Cross-Domain Communication
- System-wide DFS
- Client-Server Architecture
- Case Study: HDFS

16 July 2024

17

Client Server Model using RPC



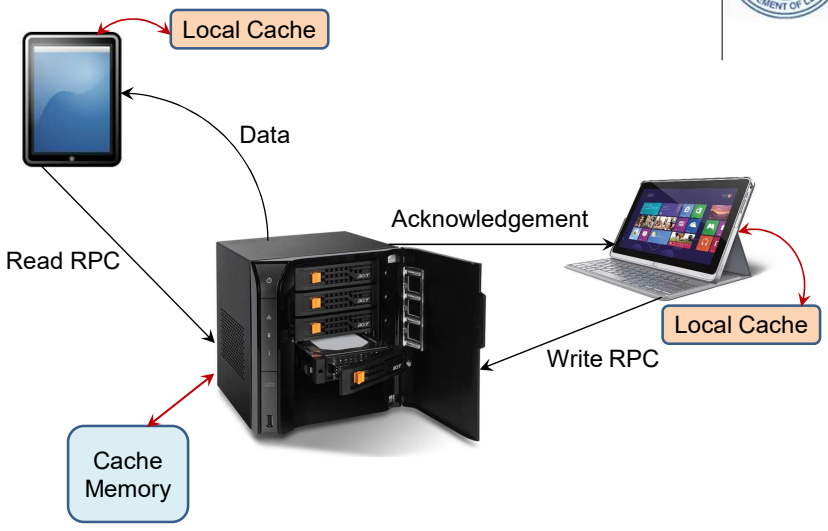
16 July 2024

18

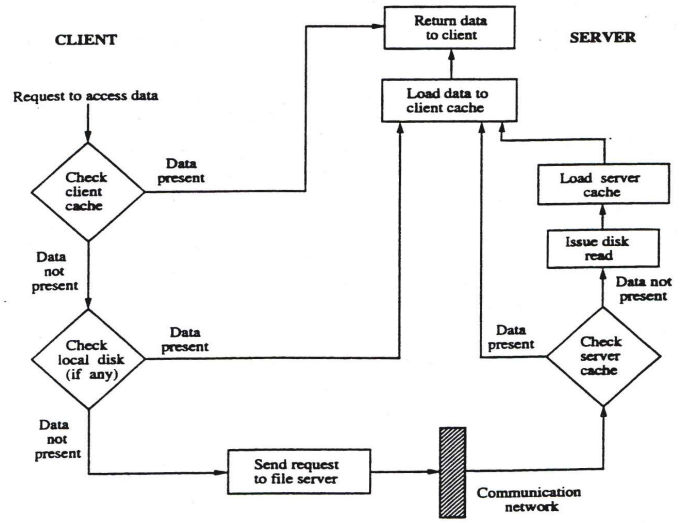
Server-Side Cache

- Reads and writes to server by RPC Calls
- Caching at server-side only
- Advantage:
  - Server provides completely consistent view of file system to multiple clients
- Concerns:
  - High latency
  - Server can be a bottleneck

Using Cache in Both Sides



Accessing DFS



Source: http://www.slideshare.net/longly/11-distributed-filesystems

Does it Solve the Problems?

- Use caching at source and destination
- Advantage:
  - Operations done locally, don't add to network traffic
  - Low latency
- Concerns:
  - Client and Server-side caches may not be mutually consistent
  - Will it be better for a Stateless server?
  - What if multiple clients access a server, some writing and some reading data?

Outline

- Cross-Domain Communication
- System-wide DFS
- Client-Server Architecture
- Case Study: HDFS

16 July 2024

23

What is HDFS?

- Hadoop Distributed File System (HDFS) is a distributed file system designed to run on a huge system built with low-cost hardware components.
  - HDFS was originally built for the Apache Nutch web search engine project.
  - HDFS later transformed into an Apache Hadoop subproject.

16 July 2024

24

Motivations…

- Using a huge number of low-cost computers for storage implies that some of these will always be non-functional.
- Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

16 July 2024

25

What's different for HDFS?

- HDFS is highly fault-tolerant and is designed for low-cost hardware.
- HDFS provides high throughput access to application data and is suitable for applications that have large data sets.
- HDFS relaxes a few POSIX (Portable Operating System Interface for UNIX) requirements for faster access to file system data.

16 July 2024

26

What's different for HDFS?

- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.
- HDFS is tuned to support large files.
- HDFS should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster.

16 July 2024

27

What's different for HDFS?

- HDFS applications assume and require a write-once-read-many access model for files.
- HDFS provides interfaces for migrating applications closer to data.
- HDFS has been designed to be easily portable from one platform to another.

16 July 2024

28

NameNode and DataNode

- HDFS has a master/slave architecture.
- An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients.
- Number of DataNodes, often one per node in the cluster, manage storage attached to the nodes that they run on.

16 July 2024

29

Role of NameNode

- The NameNode executes file system namespace operations like opening, closing, renaming files and directories.
- It also determines the mapping of blocks to DataNodes.
- NameNode is the arbitrator and repository for all HDFS metadata.

16 July 2024

30

Role of DataNode

- The DataNodes are responsible for serving read and write requests from the file system's clients.
- DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.
- DataNodes have no knowledge about HDFS files.

16 July 2024

31

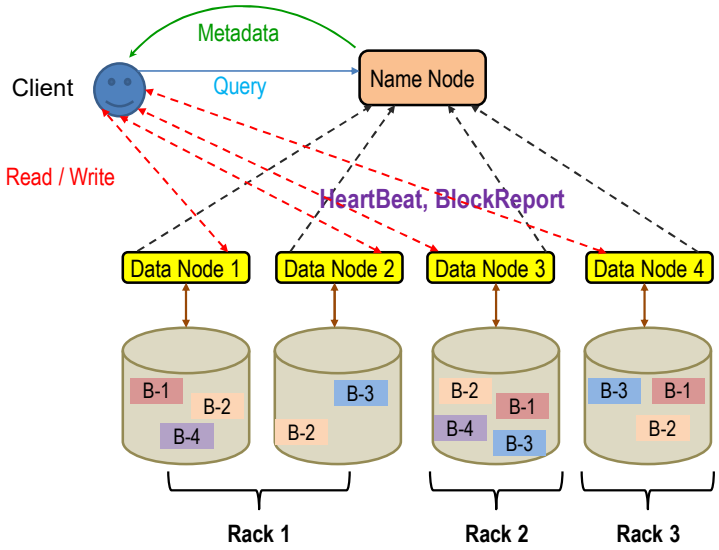
Deploying NameNode & DataNode

- In a typical deployment, there will be one dedicated machine to host only the NameNode software.
- Every other machine in the cluster runs one instance of DataNode software.
- Existence of a single NameNode in a cluster greatly simplifies architecture of the system.

16 July 2024

32

HDFS Architecture



33

Portability

- NameNode and DataNode software run on low-cost computers that may typically run a GNU/Linux OS.
- HDFS is built using Java
  - any machine that supports Java can run the NameNode or DataNode software.
  - Built on Java platform, HDFS can be deployed on a wide range of machines.

16 July 2024

34

NameNode and FS Name Space

- HDFS splits large files into smaller blocks that are stored in DataNodes.
- It is the responsibility of the NameNode to know what blocks on which DataNodes make up the complete file.
- The complete collection of all the files in the cluster is referred as the file system namespace.

16 July 2024

35

NameNode and FS Name Space

- In HDFS, the NameNode maintains the file system namespace.
- Any change to the file system namespace or its properties is recorded by the NameNode.
- HDFS supports a traditional hierarchical file organization.

16 July 2024

36

Replication in HDFS

- In HDFS, blocks of a file are replicated to improve fault tolerance.
- Files in HDFS are write-once and have strictly one writer at any time.

37

Replication Factor

- In HDFS, an application can specify the number of replicas of a file at file creation time and can be changed later
- The number of replicas to be maintained is called the replication factor of that file – by default this is 3.
- This information is also stored in the NameNode.

38

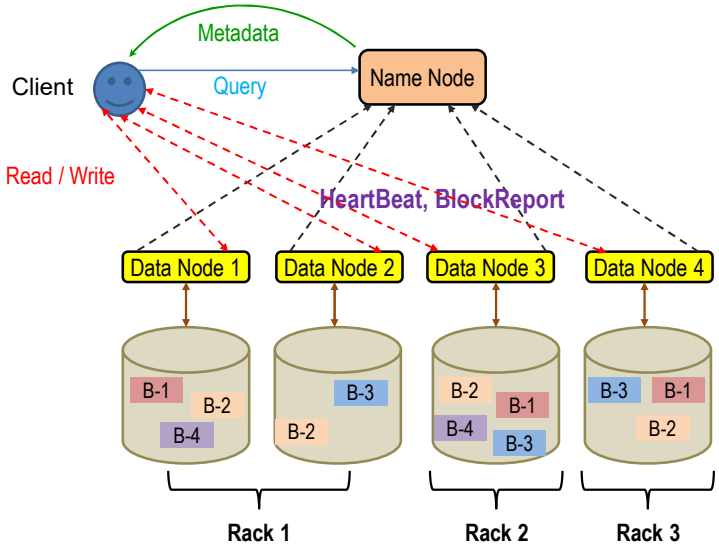
Heartbeat and Blockreport

- NameNode decides on replication of blocks
- It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster.
- Receipt of a Heartbeat implies that the DataNode is functioning properly.
- Blockreport contains a list of all blocks on a DataNode.

16 July 2024

39

HDFS Architecture



40

Replication Placement

- This makes HDFS different from others
- The purpose is to strike a balance between data reliability, availability, and network bandwidth utilization.
- HDFS often runs on a cluster of computers spread across many racks.
- Communication between nodes in different racks goes through switches.

16 July 2024

41

Replication Placement: Policy 1

- A non-optimal yet simple policy is to place replicas on unique racks.
- This prevents losing data when an entire rack fails.
- This evenly distributes replicas in the cluster and eases load balancing on component failure.
- However, cost of writes is high as a write needs to transfer blocks to multiple racks.

16 July 2024

42

Replication Placement: Policy 2

- With replication factor 3, HDFS's placement policy is to
  - put one replica on one node in the local rack,
  - another on a node in a different (remote) rack,
  - and the last on a different node in the same remote rack.
- This policy cuts the inter-rack write traffic which generally improves write performance.

16 July 2024

43

Replication Placement: Policy 2

- This policy does not impact data reliability and availability guarantees as the chance of rack failure is far less than that of node failure.
- However, it reduces the aggregate network bandwidth used when reading data since a block is placed in only two unique racks rather than 3.

16 July 2024

44

Replication Placement: Policy 2

- With this policy, the replicas of a file do not evenly distribute across the racks.
  - 1/3rd of replicas are on one node,
  - 2/3rd of replicas are on one rack, and
  - the other 1/3rd are evenly distributed across the remaining racks.
- This policy improves write performance without compromising data reliability or read performance.

16 July 2024

45

Replication Pipelining

- Suppose the replication factor is 3.
- After the client gets a list of DataNodes from the NameNode, local data is flushed to the first DataNode.
- The first DataNode
  - starts receiving the data in small sizes (4 KB)
  - writes each portion to its local repository and
  - transfers that portion to the second DataNode in the list.

16 July 2024

46

Replication Pipelining

- The second DataNode, in turn,
  - starts receiving each portion of the data block,
  - writes that portion to its repository and then
  - flushes that portion to the third DataNode.
- Finally, the third DataNode writes the data to its local repository.
- Thus, a DataNodes replicate in the pipeline.

16 July 2024

47

Safe Mode

- Initially, the NameNode enters a special state called Safe mode.
- Data blocks are not yet replicated when the NameNode is in Safe mode.
- NameNode gets Heartbeat and Blockreport messages from the DataNodes
- A block is considered safely replicated when minimum number of replicas for that data block is recorded with the NameNode

16 July 2024

48

Safe Mode

- After a configurable percentage of safely replicated data blocks checks in with the NameNode, the NameNode exits the Safe mode state.
- It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas.
- The NameNode then replicates these blocks to other DataNodes.

16 July 2024

49

EditLog and FsImage

- In HDFS, the NameNode uses a transaction log called the EditLog to record every change that occurs to FS metadata.
  - e.g., creating a new file in HDFS causes the NameNode to insert a record into the EditLog indicating this.
  - changing the replication factor of a file causes a new record to be inserted into the EditLog.
- The NameNode uses a file in its local host OS file system to store the EditLog.

16 July 2024

50

EditLog and FsImage

- The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage.
- The FsImage is stored as a file in the NameNode's local file system too.
- The NameNode keeps an image of the entire file system namespace and file Blockmap in memory.

16 July 2024

51

Checkpoint

- When the NameNode starts up:
  - it reads the FsImage and EditLog from disk
  - applies all the transactions from the EditLog to the in-memory representation of the FsImage
  - flushes out this new version into a new FsImage on disk, and
  - truncates the old EditLog.
- This process is called a checkpoint.

16 July 2024

52

DataNode and BlockReport

- A DataNode stores HDFS data in files in the local file system.
- When a DataNode starts up,
  - it scans through its local file system,
  - generates a list of all HDFS data blocks that correspond to each of these local files and
  - sends this report to the NameNode.
- This is the Blockreport.

16 July 2024

53

Data Organization

- HDFS is designed to support very large files where data is written only once but read many times at streaming speeds.
- The typical block size used by HDFS is 64 MB.
- An HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.

16 July 2024

54

Staging

- In HDFS, a client request to create a file does not reach NameNode immediately.
  - Initially, the HDFS client caches the file data into a temporary local file.
  - Application writes are transparently redirected to this temporary local file.
  - When the local file accumulates data more than one HDFS block size, the client contacts the NameNode.

16 July 2024

55

Staging

- The NameNode now inserts the file name into the file system hierarchy and allocates a data block for it.
- The NameNode responds to the client request with the identity of the DataNode and the destination data block.
- Then the client flushes the block of data from the local temporary file to the specified DataNode.

16 July 2024

56

Staging

- When a file is closed,
  - the remaining un-flushed data in temporary local file is transferred to the DataNode.
  - Client tells the NameNode that file is closed
- At this point only, the NameNode commits the file creation operation into a persistent store.
- If the NameNode dies before the file is closed, the file is lost.

16 July 2024

57

Space Reclamation

- In HDFS, a file is not immediately removed when it's deleted by the user.
- HDFS first renames it to a file in the /trash directory.
- User can Undelete a file as long as it remains in the /trash directory.
- A file remains in /trash for a configurable amount of time.

16 July 2024

58

Reducing Replication Factor

- In HDFS, when the replication factor of a file is reduced, the NameNode selects excess replicas that can be deleted.
- The next Heartbeat transfers this information to the DataNode.
- Subsequently, the DataNode removes the corresponding blocks and more free space appears in the cluster.

16 July 2024

59

Thanks for your kind attention

Questions??

60