Online Courseware

for

B.Tech. Computer Science and Engineering Program(Autonomy)

Paper Name: Distributed Operating System

Paper Code: IT(CS)605B

Online Courseware for B.Tech. Computer Science and Engineering Program(Autonomy)
Paper Name: Distributed Operating System
Paper Code: IT(CS)605B

Distributed Operating System
IT(CS)605B
Contracts: 3L
Credits- 3
Total Lecture: [33L]

Module 1:
Fundamentals of Distributed System [5]
Definition of distributed system, Examples of distributed system, Types of distributed system, Distributed Operating System, Issues in designing a distributed operating system.
System Architecture: Centralized architecture, decentralized architecture and hybrid architecture.

Communication [4]
Inter-process communication-Message Passing: features, issues, synchronization, multidatagram message, Remote Procedure Call, RPC message, Marshaling arguments and results, Server management.

Module 2:
Clock Synchronization: [2]
Physical and Logical Clock synchronization algorithms: Cristian's, Berkley, Lamport's. Global State

Distributed Mutual Exclusion:[4]
Classification of distributed mutual exclusion algorithm. Permission based: Centralized algorithm, Distributed algorithms-Ricart-Agrawala algorithm. Token based Algorithm: Suzuki-Kasami's broadcast algorithm.
Election algorithm: Bully algorithm, ring algorithm.

Distributed Deadlock Detection: [4]
Deadlock handling strategies in distributed systems. Control organizations for distributed deadlock detection. Centralized and Distributed deadlock detection algorithms: Completely Centralized algorithms, path pushing, edge chasing, global state detection algorithm.

Module 3
Distributed file systems: [6]
Issues in the design of distributed file systems: naming, transparency, update semantics and fault resilience,File Model, File accessing Models, File caching schemes, Fault Tolerance, Examples of distributed systems including Sun NFS, the Andrew filestore, CODA file system and OSF DCE.

Distributed Shared Memory: [2]
Architecture and motivations. Algorithms for implementing DSM. Memory Coherence

Module 4
Case Study: [6]
AMOEBA:Introduction, Process management, Communication.
MACH: Introduction, Process management, Communication.
DCE: Introduction, Process management, Communication.

Comparative study

Module	No. of Lecture	Existing Syllabus as per MAKAUT(Module basis)	Addition/Deletion / Comment	Justification	Syllabus for Autonomy
1	**MAKAUT(9)**	**Introduction to Distributed System [2]** Introduction, Examples of distributed system,	**Added** System Architecture: Centralized	Student will learn the deleted part in Operating	**Introduction to Distributed System [5]** Definition of distributed system, Examples of distributed system, Types of distributed

Online Courseware for B.Tech. Computer Science and Engineering Program(Autonomy)
Paper Name: Distributed Operating System
Paper Code: IT(CS)605B

	Autonomy(9)	Resource sharing, Challenges **Operating System Structures: [3]** Review of structures: monolithic kernel, layered systems, virtual machines. Process based models and client server architecture; Themicro-kernel based client-server approach. **Communication [4]** Inter-process communication, Remote Procedure Call, Remote Object Invocation, Tasks and Threads. Examples from LINUX, Solaris 2 and Windows NT.	architecture, decentralized architecture and hybrid architecture. **Deleted**Review of structures: monolithic kernel, layered systems, virtual machines. Process based models and client server architecture; The micro-kernel based client-server approach.	System.	system, Distributed Operating System, Issues in designing a distributed operation system. System Architecture: Centralized architecture, decentralized architecture and hybrid architecture. **Communication [4]** Inter-process communication-Message Passing: features, issues, synchronization, multidatagram message, Remote Procedure Call, RPC message, Marshaling arguments and results, Server management.
2	**MAKAUT(10)** **Autonomy(10)**	**Theoretical Foundations: [2]** Introduction. Inherent Limitations of distributed Systems. Lamport's Logical clock. Global State **Distributed Mutual Exclusion:[4]** Classification of distributed mutual exclusion algorithm. NonToken based Algorithm:Lamport's algorithm, Ricart-Agrawala algorithm. Token based Algorithm: Suzuki-Kasami's broadcast algorithm. **Distributed Deadlock Detection: [4]** Deadlock handling strategies in distributed systems. Control organizations for distributed deadlock detection. Centralized and Distributed deadlock detection algorithms: Completely Centralized algorithms, path pushing, edge chasing, global state detection algorithm.	**Added** Election algorithms are introduced	Those algorithms are important to learn the mutual exclusion and deadlock.	**Theoretical Foundations: [2]** Introduction. Inherent Limitations of distributed Systems. Clock synchronization, Lamport's Logical clock. Global State **Distributed Mutual Exclusion:[4]** Classification of distributed mutual exclusion algorithm. Centralized algorithm, Ricart-Agrawala algorithm. Token based Algorithm: Suzuki-Kasami's broadcast algorithm. Election algorithm: Bully algorithm, ring algorithm. **Distributed Deadlock Detection: [4]** Deadlock handling strategies in distributed systems. Control organizations for distributed deadlock detection. Centralized and Distributed deadlock detection algorithms: Completely Centralized algorithms, path pushing, edge chasing, global state detection algorithm.
3	**MAKAUT(8)** **Autonomy(8)**	**Distributed file systems: [6]** Issues in the design of distributed file systems: naming, transparency, update semantics and fault resilience. Use of the Virtual File System layer. Examples of distributed systems including Sun NFS, the Andrew file store, CODA file system and OSF DCE. **Distributed Shared Memory: [2]** Architecture and motivations. Algorithms for implementing DSM. Memory Coherence	-	-	**Distributed file systems: [6]** Issues in the design of distributed file systems: naming, transparency, update semantics and fault resilience. Use of the Virtual File System layer. Examples of distributed systems including Sun NFS, the Andrew file store, CODA file system and OSF DCE. **Distributed Shared Memory: [2]** Architecture and motivations. Algorithms for implementing DSM. Memory Coherence
4	**MAKAUT(7)** **Autonomy(6)**	**Protection and Security: [4]** Requirements for protection and security regimes. The access matrix model of protection. System and user modes, rings of protection, access lists, capabilities. User authentication, passwords and signatures. Use of single key and public key encryption. **CORBA: [3]** The Common Object Request Broker Architecture model and software and its relationship to Operating Systems.	**Deleted** Protection and security and CORBA **Added** Case study	CORBA is obsolete so it is not required to study. Instead student can learn some existing system	**Case Study: [6]** AMOEBA:Introduction, Process management, Communication. MACH: Introduction, Process management, Communication. DCE: Introduction, Process management, Communication.

Module 1: Fundamentals of Distributed System

Lecture 1: Definition and examples

Distributed computing is the process of aggregating the power of several computing entities to collaboratively run a computational task in a transparent and coherent way, so that it appears as a single, centralized system.

Important characteristics of distributed systems are:

- A distributed system consists of components (i.e., computers) that are autonomous.
- Users (be they people or programs) think they are dealing with a single system. In spite of having the differences between the various computers and the ways in which they communicate, are mostly hidden from users. The same holds for the internal organization of the distributed system.
- Another important characteristic is that users and applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place.
- In principle, distributed systems should also be relatively easy to expand or scale. This characteristic is a direct consequence of having independent computers, but at the same time, hiding how these computers actually take part in the system as a whole.
- A distributed system will normally be continuously available, although perhaps some parts may be temporarily out of order. Users and applications should not notice that parts are being replaced or fixed, or that new parts are added to serve more users or applications.

➢ Middleware:

In order to support heterogeneous computers and networks while offering a single-system view, distributed systems are oftenorganized by means of alayer of software-that is, logically placed between a higher-level layer consisting of users andapplications, and a layer underneath consisting of operating systems and basic communication facilities, as shown in Fig. 1-1 Accordingly, such a distributed systemis sometimes called middleware.

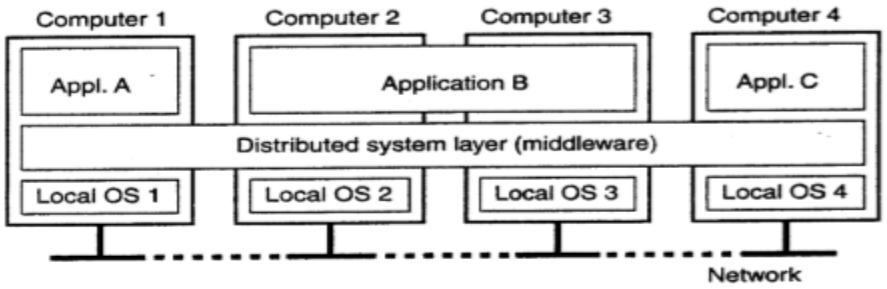
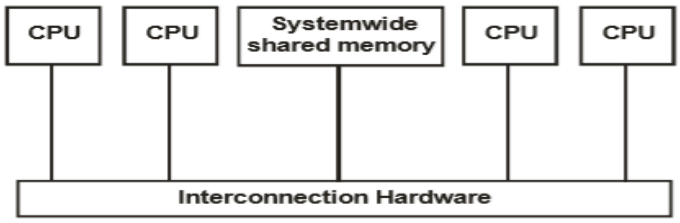


Figure I-I. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

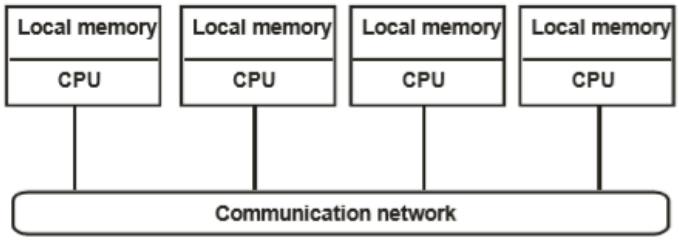
Fig. 1-1shows four networked computers and three applications, of which application B is distributed across computers 2and 3.Each application, is offered the same interface. The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate. At the same time, it hides, as best and reasonable as possible, the differences in hardware and operating systems from each application.

Before going into the actual details of the distributed systems let us study how distributed computing evolved.

- **Computer architectures consisting of interconnected, multiple processors are basically of two types:**
    1. **Tightly coupled systems:** In these systems, there is a single system wide primary memory (address space) that is shared by all the processors. 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.



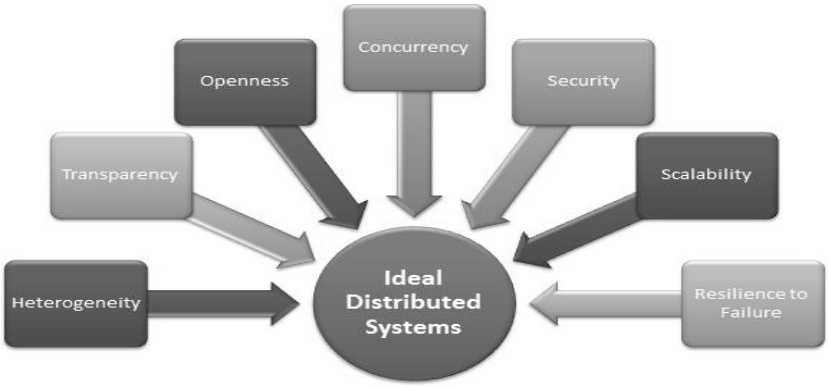
    2. **Loosely coupled systems:** In these systems, the processors do not share memory, and each processor has its own local memory. If a processor writes the value 100 to the memory location x, this write operation will only change the contents of its local memory and will not affect the contents of the memory. In these systems, all physical communication between the processors is done by passing messages across the network that interconnects the processors.



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.

Lecture 2 & 3: Issues in designing a distributed operating system

Design goals of distributed systems



In order to design a good distributed system, there are six key design goals. They are:

- **Concurrency**
- **Scalability**
- **Openness**
- **Fault Tolerance**
- **Privacy and Authentication**
- **Transparency.**

➢ Concurrency:-

A server must handle many client requests at the same time. Distributed systems are naturally concurrent; that is, there are multiple workstations running programs independently and at the same time. Concurrency is important because any distributed service that isn't concurrent would become a bottleneck that would serialize the actions of its clients and thus reduce the natural concurrency of the system.

➢ Scalability:-

The goal is to be able to use the same software for different size systems. A distributed software system is scalable if it can handle increased demand on any part of the system (i.e., more clients, bigger networks, faster networks, etc.) without a change to the software. In other words, we would like the engineering impact of increased demand to be proportional to that increase. Distributed systems, however, can be built for a very wide range of scales and it is thus not a good idea to try to build a system that can handle everything. A local-area network file server should be built differently from a Web server that must handle millions of requests a day from throughout the world. The key goal is to understand the target system's expected size and expected growth and to understand how the distributed system will scale as the system grows.

- **Openness:-**

Two types of openness are important: non-proprietary and extensibility. Public protocols are important because they make it possible for many software manufacturers to build clients and servers that will be able to talk to each other. Proprietary protocols limit the "players" to those from a single company and thus limit the success of the protocol. A system is extensible if it permits customisations needed to meet unanticipated requirements. Extensibility is important because it aids scalability and allows a system to survive over time as the demands on it and the ways it is used change.

- **Fault Tolerance:-**

It is critically important that a distributed system be able to tolerate "partial failures". Why is it so important? Two reasons are as follows:

• Failures are more harmful: Many clients are affected by the failure of a distributed service, unlike a non-distributed system in which a failure affects only a single node.

• Failures are more likely: A distributed service depends on many components (workstation nodes, network interfaces, networks, switches, routers, etc.) all of which must work. Furthermore, a client will often depend on multiple distributed services (e.g., multiple file systems or databases) in order to function properly. The probability that such a client will experience a failure can be approximated as the sum of the individual failure probabilities of everything that it depends on. Thus, a client that depends on N components (hardware or software) that each have failure probability P will fail with probability roughly N*P. (This approximation is valid for small values of P. The exact failure probability is (1-(1-P)^N).)

There are two aspects of failure tolerance to be studied as shown below:

- **Recovery**

• A failure shouldn't cause the loss (or corruption) of critical data, or computation.

• After a failure, the system should recover critical data, even data that was being modified when the failure occurred. Data that survives failures is called "persistent" data.

• Very long-running computations must also be made recoverable in order to restart them where they left off instead of from the beginning.

• For example, if a fileserver crashes, the data in the file system it serves should be intact after the server is restarted.

- **Availability**

• A failure shouldn't interrupt the service provided by a critical server.

• This is a bit harder to achieve than recovery. We often speak of a highlyavailable service as one that is almost always available even if failures occur.

• The main technique for ensuring availability is service replication.

• For example, a fileserver could be made highly available by running two copies of the server on different nodes. If one of the servers fails, the other should be able to step in without service interruption.

- **Privacy and Authentication:-**

Privacy is achieved when the sender of a message can control what other programs (or people) can read the message. The goal is to protect against eavesdropping. For example, if you use your credit card to buy something over the Web, you will probably want to prevent anyone but the target Web server from reading the message that contains your credit card account number. Authentication is the process of ensuring that programs can know who they are talking to. This is important for both clients and servers.

For clients authentication is needed to enable a concept called trust. For example, the fact that you are willing to give your credit card number to a merchant when you buy something means that you are implicitly trusting that merchant to use your number according to the rules to which you have both agreed (to debit your account for the amount of the purchase and give the number to no one else). To make a Web purchase, you must trust the merchant's Web server just like you would trust the merchant for an in-person purchase. To establish this trust, however, you must ensure that your Web browser is really talking to the merchant's Web server and not to some other program that's just pretending to be their merchant.

For servers authentication is needed to enforce access control. For a server to control who has access to the resources it manages (your files if it is a fileserver, your money if it is a banking server), it must know who it is talking to. A Unix login is a crude example of an authentication used to provide access control. It is a crude example because a remote login sends your username and password in messages for which privacy is not guaranteed. It is thus possible, though usually difficult, for someone to eavesdrop on those messages and thus figure out your username and password.

For a distributed system, the only way to ensure privacy and authentication is by using cryptography.

- **Transparency:-**

The final goal is transparency. We often use the term single system image to refer to this goal of making the distributed system look to programs like it is a tightly coupled (i.e., single) system.

This is really what a distributed system software is all about. We want the system software (operating system, runtime library, language, compiler) to deal with all of the complexities of distributed computing so that writing distributed applications is as easy as possible.

Achieving complete transparency is difficult. There are eight types, namely:

• ***Access Transparency***: enables local and remote resources to be accessed using identical operations

• ***Location Transparency:*** enables resources to be accessed without knowledge of their (physical) location. Access transparency and location transparency are together referred to as network transparency.

• ***Concurrency Transparency:*** enables several processes to operate concurrently using shared resources without interference between them.

• ***Replication Transparency:*** enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.

• ***Failure Transparency:*** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.

• ***Mobility Transparency:*** allows the movement of resources and clients within a system without affecting the operation of users or programs.

• ***Performance Transparency:*** allows the system to be reconfigured to improve performance as loads change.

• ***Scaling Transparency:*** Transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

Lecture 4: System Architecture

Design issues involved in distributed systems

There are certain design issues to be considered for distributed systems. They are:
a) Naming
b) Communication
c) Software Structure
d) Workload Allocation
e) Consistency Maintenance

a) Naming

A name is a string of characters used to identify and locate a distributed resource. An identifier is a special kind of name that is used directly by the computer to access the resource. For example the identifier for a Unix server would include at least (1) an IP address and (2) a port number. The IP address is used to find the node that runs the server and the port number identifies the server process on that node.

Resolution is the process of turning a name into an identifier. Resolution is performed by a Name Server. It is also called "binding" (as in binding a name to a distributed service). For example, an IP domain name (e.g., cs.ubc.ca) is turned into an IP address by the IP Domain Name Server (DNS), a distributed hierarchical server running in the Internet.

b) Communication

Getting different processes to talk to each other through Messages or Remote method invocation.

c) Software Structure

• The main issues are to choose a software structure that supports our goals, particularly the goal of openness.

• We thus want structures that promote extensibility and otherwise make it easy to program the system.

• Alternatives are:

- A monolithic structure is basically a big pile of code; it is not so desirable because it is hard to extend or reason about a system like that.

- A modular structure divides the system into models with well-defined interfaces that define how the models interact. Modular systems are more extensible and easier to reason about than monolithic systems.
- A layered structure is a special type of modular structure in which modules are organised into layers (one on top of the other). The interaction between layers is restricted such that a layer can communicate directly with only the layer immediately above and the layer immediately below it. In this way, each layer defines an abstraction that is used by the layer immediately above it. Clients interact only with the top layer and only the bottom layer deals with the hardware (e.g., network, disk, etc.) Network protocols have traditionally been organised as layers (as we will see in the next class) and for this reason we often refer to these protocols as "protocol stacks". o Operating systems can be either monolithic (e.g., UNIX and Windows) or modular (e.g., Mach and Windows NT). A modular operating system is called a micro kernel. A micro-kernel OS has a small minimalist kernel that includes as little functionality as possible. OS services such as VM, file systems, and networking are added to the system as separate servers and they reside in their own user-mode address spaces outside the kernel.

d) Workload Allocation

• The key issue is load balancing: The allocation of the network workload such that network resources (e.g., CPUs, memory, and disks) are used efficiently.

For CPUs, there are two key approaches. They are:

- Processor Pools
- Idle Workstations

e) Consistency Maintenance

The final issue is consistency. There are four key aspects of consistency: atomicity, coherence, failure consistency, and clock consistency.

Advantages and disadvantages of distributed system over centralized system

Advantages:

- **Incremental growth:** Computing power can be added in small Increments.
- **Reliability:** If one machine crashes, the system as a whole can still survive.
- **Speed:** A distributed system may have more total computing power than a mainframe.

- **Open system:** This is the most important point and themost characteristic point of a distributed system. Since it is an open system it is always ready to communicate with other systems. An open system that scales has an advantage over a perfectly closed and self-contained system.
- **Economic:** Microprocessors offer a better price/performance than mainframes.

Disadvantages:

- Security problem due to sharing.
- Some messages can be lost in the network system.
- Bandwidth is another problem if there is large data then all network wires to be replaced which tends to become expensive.
- Overloading is another problem in distributed operating systems.
- If there is a database connected on local system and many users accessing that database through remote or distributed way then performance become slow.
- The databases in network operating is difficult to administrate then single user system.

Examples of distributed operating systems:-

- Windows server 2003
- Windows server 2008
- Windows server 2012
- Ubuntu
- Linux (Apache Server)

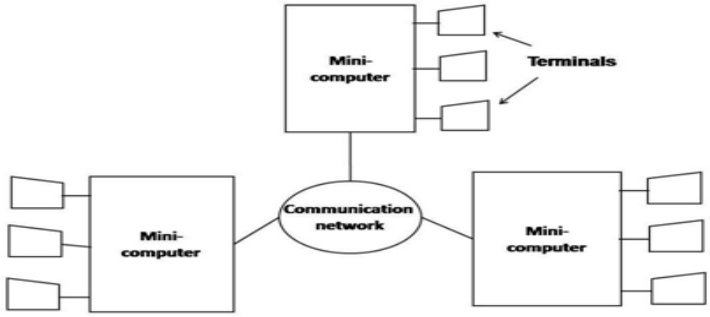
Lecture: 5: System Models

Distributed Computing System Models

Distributed Computing system models can be broadly classified into five categories. They are

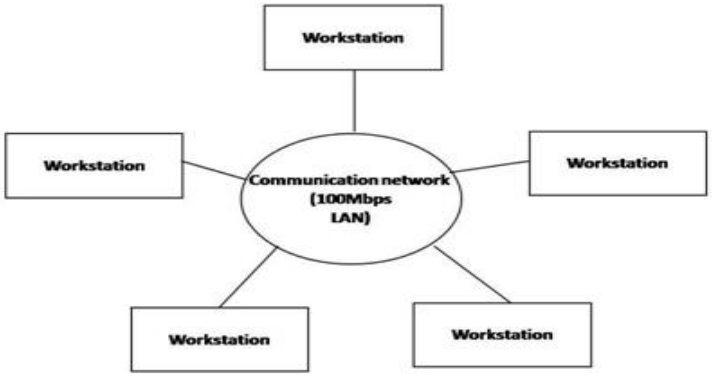
- ✓ **Minicomputer model**
- ✓ **Workstation model**
- ✓ **Workstation – server model**
- ✓ **Processor – pool model**
- ✓ **Hybrid model**

Minicomputer Model



The minicomputer model is a simple extension of the centralized time-sharing system. A distributed computing system based on this model consists of a few minicomputers (they may be large supercomputers as well) interconnected by a communication network. Each minicomputer usually has multiple users simultaneously logged on to it. For this, several interactive terminals are connected to each minicomputer. Each user is logged on to one specific minicomputer, with remote access to other minicomputers. The network allows a user to access remote resources that are available on some machine other than the one on to which the user is currently logged. The minicomputer model may be used when resource sharing (such as sharing of information databases of different types, with each type of database located on a different machine) with remote users is desired. The early ARPAnet is an example of a distributed computing system based on the minicomputer model.

Workstation Model

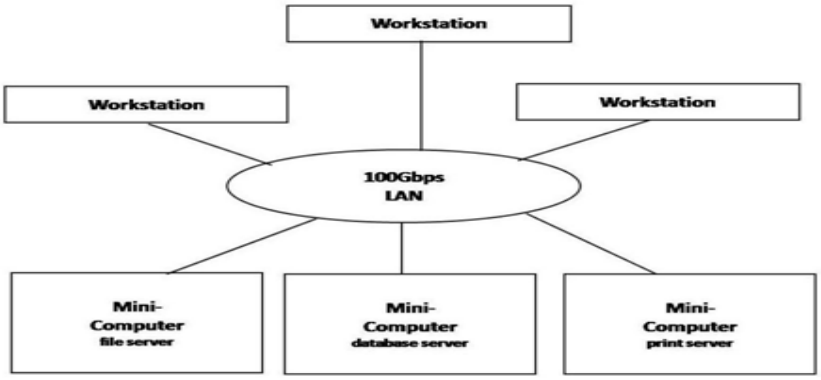


A distributed computing system based on the workstation model consists of several workstations interconnected by a communication network. An organization may have several workstations located throughout a building or campus, each workstation equipped with its own disk and serving as a single-user computer. It has been often found that in such an environment, at any one time a significant proportion of the workstations are idle (not being used), resulting in the waste of large amounts of CPU time. Therefore, the idea of the workstation model is to interconnect all these workstations by a high-speed LAN so that idle workstations may be used to process jobs of users who are logged onto other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently. Example:Sprite system & Xerox PARC.

Workstation – Server Model

The workstation model is a network of personal workstations, each with its own disk and a local file system. A workstation with its own local disk is usually called a diskful workstation and a workstation without a local disk is called a diskless workstation. With the proliferation of high-speed networks, diskless workstations have become more popular in network environments than diskful workstations, making the workstation-server model more popular than the workstation model for building distributed computing systems.

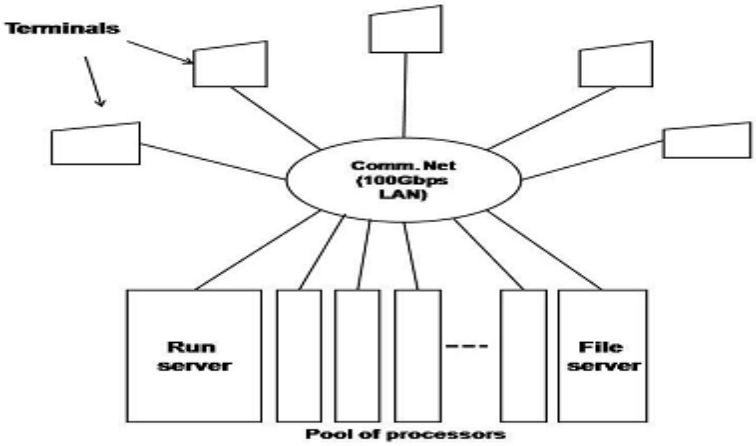
A distributed computing system based on the workstation-server model consists of a few minicomputers and several workstations (most of which are diskless, but a few of which may be diskful) interconnected by a communication network.



Note that when diskless workstations are used on a network, the file system to be used by these workstations must be implemented either by a diskful workstation or by a minicomputer equipped with a disk for file storage. One or more of the minicomputers are used for implementing the file system. Other minicomputers may be used for providing other types of services, such as database service and print service. Therefore, each minicomputer is used as a server machine to provide one or more types of services. Therefore in the workstation-server model, in addition to the workstations, there are specialized machines (may be specialized workstations) for running server processes (called servers) for managing and providing access to shared resources. For a number of reasons, such as higher reliability and better scalability, multiple servers are often used for managing the resources of a particular type in a distributed computing system. For example, there may be multiple file servers, each running on a separate minicomputer and cooperating via the network, for managing the files of all the users in the system. Due to this reason, a distinction is often made between the services that are provided to clients and the servers that provide them. That is, a service is an abstract entity that is provided by one or more servers. For example, one or more file servers may be used in a distributed computing system to provide file service to the users.

In this model, a user logs onto a workstation called his or her home workstation. Normal computation activities required by the user's processes are performed at the user's home workstation, but requests for services provided by special servers (such as a file server or a database server) are sent to a server providing that type of service that performs the user's requested activity and returns the result of request processing to the user's workstation. Therefore, in this model, the user's processes need not migrated to the server machines for getting the work done by those machines.

Processor – Pool Model



The processor-pool model is based on the observation that most of the time a user does not need any computing power but once in a while the user may need a very large amount of computing power for a short time (e.g., when recompiling a program consisting of a large number of files after changing a basic shared declaration). Therefore, unlike the workstation-server model in which a processor is allocated to each user, in the processor-pool model the processors are pooled together to be shared by the users as needed. The pool of processors consists of a large number of microcomputers and minicomputers attached to the network. Each processor in the pool has its own memory to load and run a system program or an application program of the distributed computing system.

Hybrid Model

Out of the four models described above, the workstation-server model, is the most widely used model for building distributed computing systems. This is because a large number of computer users only perform simple interactive tasks such as editing jobs, sending electronic mails, and executing small programs. The workstation-server model is ideal for such simple usage. However, in a working environment that has groups of users who often perform jobs needing massive computation, the processor-pool model is more attractive and suitable.

Lecture 6: Inter-process communication

A distributed system is basically a computer network whose nodes has their own local memory and may also have other hardware and software resources. A distributed system, therefore, relies entirely on the underlying computer network for the communication of data and control information between the nodes of which they are composed. Furthermore, the performance and reliability of a distributed system depend to a great extent on the performance and reliability of the underlying computer network.

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.

Inter process 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:** 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.

2. **Copy sharing, or message-passing approach:** 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).

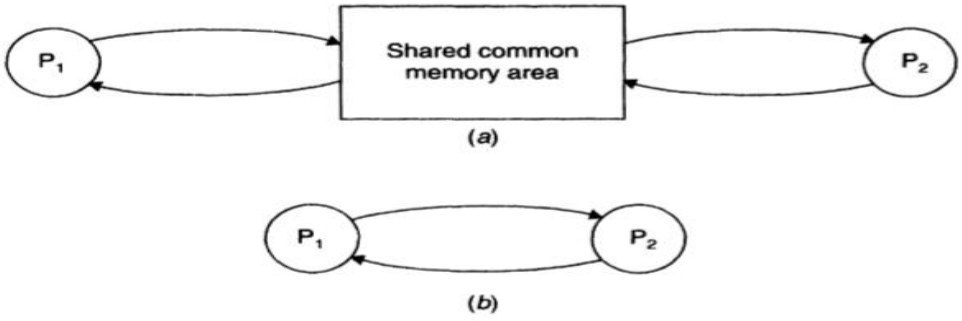


Fig. The two basic interprocess communication paradigms: (*a*) The shared-data approach. (*b*) The message-passing approach.

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.

Features of a good message passing system

- **Simplicity:** A message-passing system should be easy to use, easy to develop new applications that communicate with the existing ones, able to hide the details of underlying network protocols used.
- **Efficiency** :
    1. It can be made efficient by reducing number of message exchange.
    2. Avoiding cost of setting and terminating connections between the same pair.
    3. Minimizing the cost of maintaining connections.
    4. Piggybacking.

- **Uniform Semantics:** In a distributed system, a message-passing system may be used for the following two types of inter process 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.

• **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.

- **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:

    i. Atomicity
    ii. Ordered delivery
    iii. 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.

- **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:

    i. Authentication of the receiver(s) of a message by the sender.
    ii. Authentication of the sender of a message by its receiver(s).
    iii. Encryption of a message before sending it over the network.

- **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.

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. 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.
- **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.

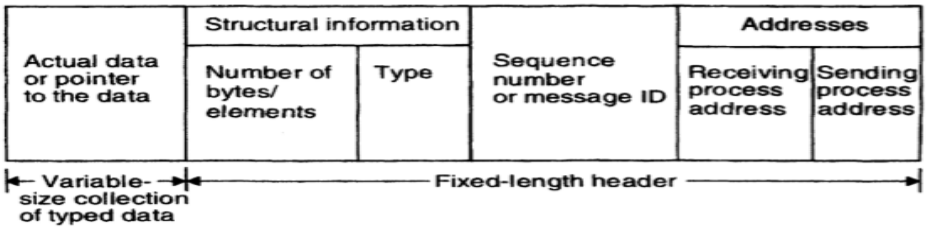


Fig. A typical message structure.

Lecture 7: Synchronization

The semantics used for synchronization may be broadly classified as blocking and non blocking types. A primitive is said to have non blocking semantics if its invocation does not block the execution of its invoker, otherwise a primitive is said to be of the blocking type. The synchronization imposed on the communicating processes basically depends on one of the two types of semantics used for the send and receive primitives.

In case of a blocking send primitive, after execution of the send statement, the sending process is blocked until it receives an acknowledgment from the receiver that the message has been received. In the case of a blocking receive primitive, after execution of the receive statement; the receiving process is blocked until it receives a message.

On the other hand, for non blocking send primitive, after execution of the send statement, the sending process is allowed to proceed with its execution as soon as the message has been copied to a buffer. For a non-blocking receive primitive, the receiving process proceeds with its execution after execution of the receive statement, which returns control almost immediately just after telling the kernel where the message buffer is.

An important issue in a non-blocking receive primitive is how the receiving process knows that the message has arrived in the message buffer. One of the following two methods is commonly used for this purpose:

1**. Polling:** In this method, a test primitive is provided to allow the receiver to check the buffer status. The receiver uses this primitive to periodically poll the kernel to check if the message is already available in the buffer.

2. **Interrupt:** In this method, when the message has been filled in the buffer and is ready for use by the receiver, a software interrupt is used to notify the receiving process. This method permits the receiving process to continue with its execution without having to issue unsuccessful test requests. Although this method is highly efficient and allows maximum parallelism, its main drawback is that user-level interrupts make programming difficult.

In a blocking send primitive, the sending process could get blocked forever in situations where the potential receiving process has crashed or the sent message has been lost on the network due to communication failure. To prevent this situation, blocking send primitives often use a timeout value. A timeout value may also be associated with a blocking receive primitive to prevent the receiving process from getting blocked indefinitely in situations where the potential sending process has crashed or the expected message has been lost on the network due to communication failure.

When both the send and receive primitives of a communication between two processes use blocking semantics, the communication is said to be synchronous; otherwise it is asynchronous.

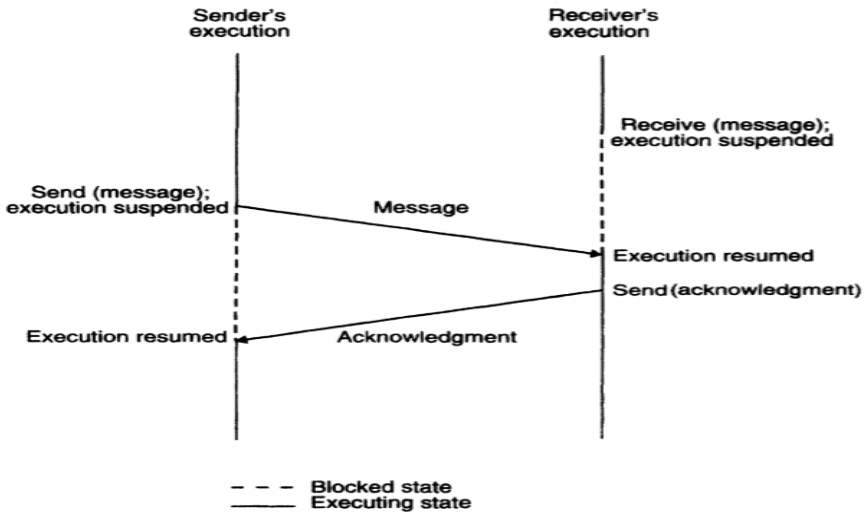


Fig. Synchronous mode of communication with both *send* and *receive* primitives having blocking-type semantics.

As compared to asynchronous communication, synchronous communication is simple and easy to implement. It also contributes to reliability because it assures the sending process that its message has been accepted before the sending process resumes execution. However, the main drawback of synchronous communication is that it limits concurrency and is subject to communication deadlocks.

Multidatagram Messages

Almost all networks have an upper bound on the size of data that can be transmitted at a time. This size is known as the maximum transfer unit (MTU) of a network. A message whose size is greater than the MTU has to be fragmented into multiples of the MTU, and then each fragment has to be sent separately. Each fragment is sent in a packet that has some control information in addition to the message data. Each packet is known as a datagram. Messages smaller than the MTU of the network can be sent in a single packet and are known as single-datagram messages. On the other hand, messages larger than the MTU of the network have to be fragmented and sent in multiple packets. Such messages are known as multidatagram messages. Obviously, different packets of a multidatagram message bear a sequential relationship to one another. The disassembling of a multidatagram message into multiple packets on the sender side and the reassembling of the packets on the receiver side is usually the responsibility of the message-passing system.

Lecture 8: Remote Procedure Call

Remote Procedure Call (RPC) provides a different paradigm for accessing network services. Instead of accessing remote services by sending and receiving messages, a client invokes services by making a local procedure call. The local procedure hides the details of the network communication.

When making a remote procedure call:

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.
2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.

The main goal of RPC is to hide the existence of the network from a program. As a result, RPC doesn't quite fit into the OSI model:

1. The message-passing nature of network communication is hidden from the user. The user doesn't first open a connection, read and write data, and then close the connection. Indeed, a client often doesn’t even know they are using the network!
2. RPC often omits many of the protocol layers to improve performance. Even a small performance improvement is important because a program may invoke RPCs often. For example, on (diskless) Sun workstations, every file access is made via an RPC.

RPC is especially well suited for client-server (e.g., query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

The following steps take place during an RPC:

1. A client invokes a *client stub* procedure, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub *marshalls* the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.
3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a *server stub*, which demarshalls the parameters and calls the desired server routine using the regular procedure call mechanism.

5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

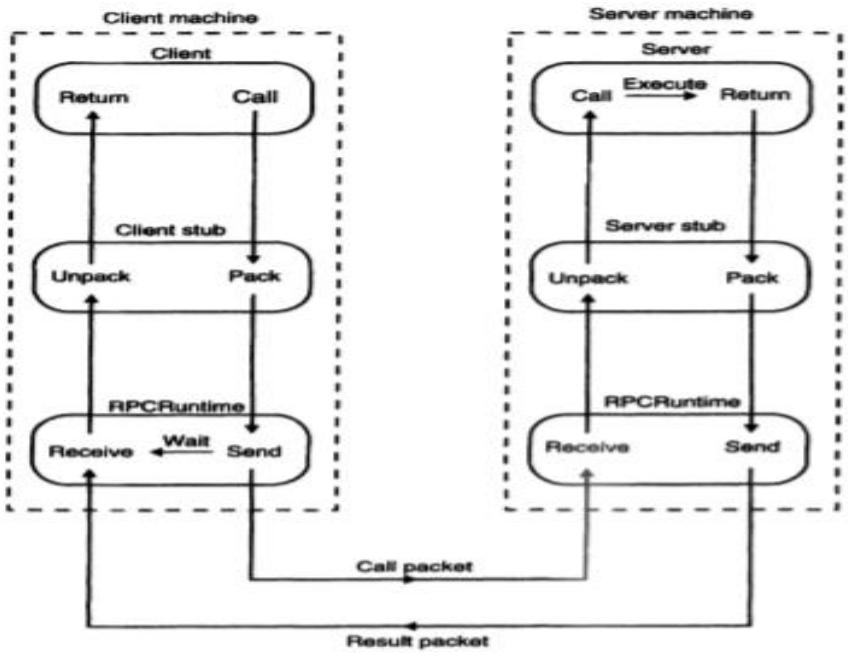


Fig. 4.2 Implementation of RPC mechanism.

RPC Operations:

1) Conventional procedure call: For a call of a program, an empty stack is present to make the call, the caller pushes the parameters onto the stack (last one first order). After the read has finished running, it puts the return values in a register and removes the return address and transfers controls back to the caller. Parameters can be called by value or reference.

- ✓ **Call by Value:** Here the parameters are copied into the stack. The value parameter is just an initialized local variable. The called procedure may modify the variable, but such changes do not affect the original value at the calling side.
- ✓ **Call by reference:** It is a pointer to the variable. In the call to Read, the second parameter is a reference parameter. It does not modify the array in the calling procedure.
- ✓ **Call-by-copy:** Another parameter passing mechanism exists along with the above two, it's called call-by-copy or Restore. Here the caller copies the variable to the stack and then copies the variable to the stack and then

copies it back after the call, overwriting the caller's original values. The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language. Sometimes it depends on the data type being passed.

2) Client and Server Stubs: A stub in distributed computing is a piece of code used for converting parameters passed during a Remote Procedure Call.

 The main idea of an RPC is to allow a local computer (client) to remotely call procedures on a remote computer (server). The client and server use different address spaces, so conversion of parameters used in a function call have to be performed; otherwise the values of those parameters could not be used, because of pointers to the computer's memory pointing to different data on each machine.

 The client and server may also use different data representations even for simple parameters. Stubs are used to perform the conversion of the parameters, so a Remote Function Call looks like a local function call for the remote computer.

For transparency of RPC, the calling procedure should not know that the called procedure is executing on a different machine.

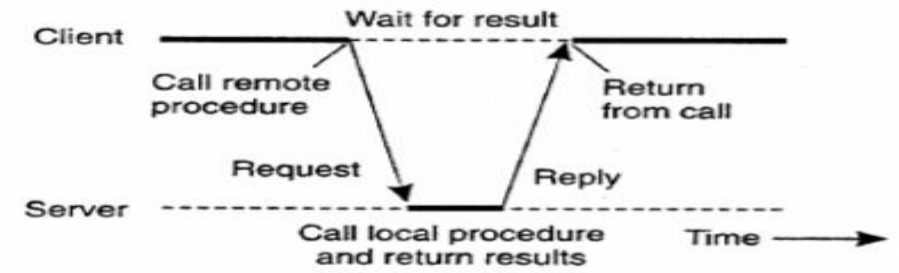


Figure 3.1: Principle of RPC between a client and server program.

 Client Stub: Used when read is a remote procedure. Client stub is put into a library and is called using a calling sequence. It calls for the local operating system. It does not ask for the local operating system to give data, it asks the server and then blocks itself till the reply comes.

 Server Stub: when a message arrives, it directly goes to the server stub. Server stub has the same functions as the client stub. The stub here unpacks the parameters from the message and then calls the server procedure in the usual way.

Summary of the process:

1) The client procedure calls the client stub in the normal way.
2) The client stub builds a message and calls the local operating system.

3) The client's as sends the message to the remote as.
4) The remote as gives the message to the server stub.
5) The server stub unpacks the parameters and calls the server.
6) The server does the work and returns the result to the stub.
7) The server stub packs it in a message and calls its local as.
8) The server's as sends the message to the client's as.
9) The client's as gives the message to the client stub.
10) The stub unpacks the result and returns to the client.

RPC messages

Any remote procedure call involves a client process and a server process that are possibly located on different computers. 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.

Call Messages

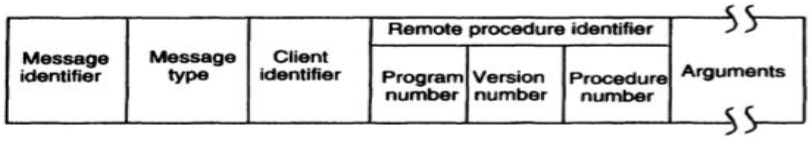


Fig. 4.3 A typical RPC call message format.

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:

1. A message identification field that consists of a sequence number.
2. A message type field that is used to distinguish call messages from reply messages.
3. 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.

Reply Messages

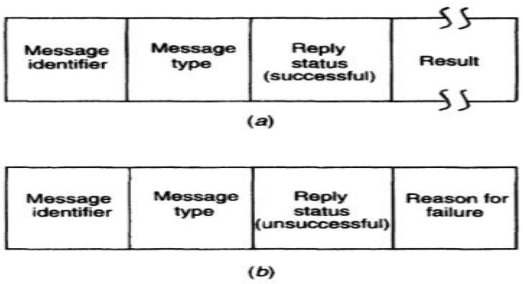


Fig. 4.4 A typical RPC reply message format: (*a*) a successful reply message format; (*b*) an unsuccessful reply message format.

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 1or 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.

Lecture 9: Marshaling arguments and results

- Marshalling is the packing of procedure parameters into a message packet.
- The RPC stubs call type-specific procedures to marshall (or unmarshall) all of the parameters to the call.
- On the client side, the client stub marshalls the parameters into the call packet; on the server side the server stub unmarshalls the parameters in order to call the server's procedure.
- On the return, the server stub marshalls return parameters into the return packet; the client stub unmarshalls return parameters and returns to the client.

Actions involved in marshalling:

1. First, an application issues an invocation request by locally calling the associated method, just like calling a procedure in an RPC.
2. The control sub object checks the user permissions with the information stored in the local security object. In this case, the security object should have a valid user certificate.
3. The request is marshaled and passed on.
4. The replication sub object requests the middleware to set up a secure channel to a suitable replica.
5. The security object first initiates a replica lookup. To achieve this goal, it could use any naming service that can look up replicas that have been specified to be able to execute certain methods. The Globe location service has been modified to handle such lookups.
6. Once a suitable replica has been found, the security sub object can set up a secure channel with its peer, after which control is returned to the replication sub object. Note that part of this establishment requires that the replica proves it is allowed to carry out the requested invocation.
7. The request is now passed on to the communication sub object.
8. The sub object encrypts and signs the request so that it can pass through the channel.
9. After its receipt, the request is decrypted and authenticated.
10. The request is then simply passed on to the server-side replication sub object.
11. Authorization takes place: in this case the user certificate from the client-side stub has been passed to the replica so that we can verify that the request can indeed be carried out.
12. The request is then un-marshaled.
13. Finally, the operation can be executed.

Server management

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

Server Implementation:

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

- **Stateful Servers**

  - Client's state information from one RPC to the other is maintained within a Stateful server.
  - When two subsequent calls are made by a client to the Stateful servers, some state information of the service performed for the client is stored by server process, which is used at while executing the next call.

In a server for byte stream files the following operations take place:

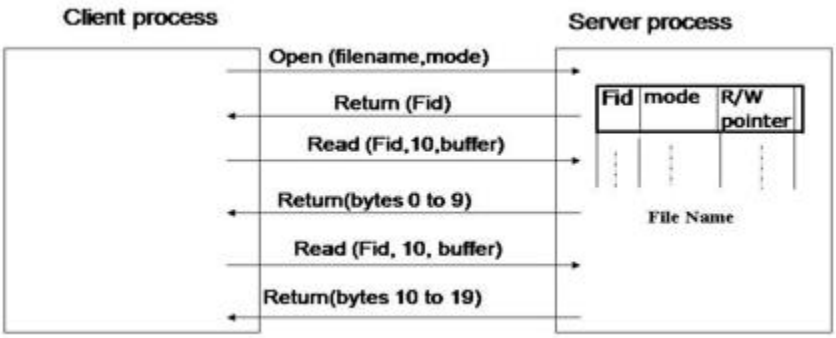
i. Open (filename, mode): This operation opens a file named filename in the specified mode. An entry for this file in file table is created which maintains file state information. When file is opened R/W pointer is set to zero and the client receives the file Fid.

ii. Read (Fid, m, buffer): This operation gets m bytes of data from the Fid file into the buffer. When this operation is executed the client receives m bytes of file data starting from the byte addressed by R/W pointer and then the pointer is incremented by m.

iii. Write (Fid, m, buffer): When this operation is executed, m bytes of data are taken from the specified buffer and writes it into the Fid file at byte position which is addressed by the W/R pointer and then increments the pointer by m.

iv. Seek (Fid, position): This operation changes the value of read-write pointer of the file Fid to a new value specified as position.

v. Close (Fid): This operation is used to delete file state information of the file Fid from its file-table.



Example of Stateful file server

- For the above diagram after opening a file if a client makes to subsequent Read(Fid,10,buffer) calls, the first call will return first 10 bytes(0-9) and the second call will return the next 10 bytes(10-19).

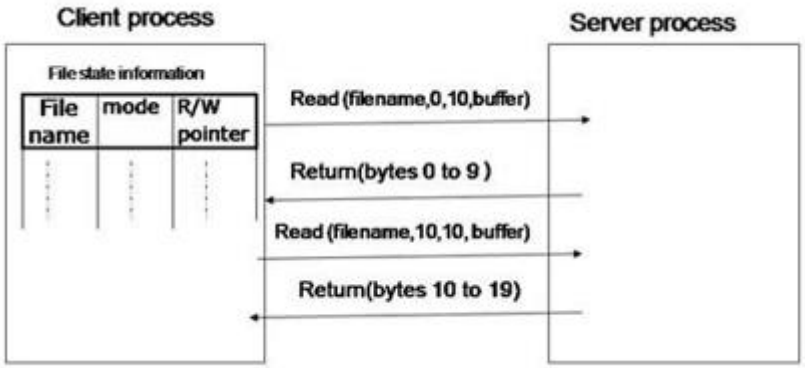
- **Stateless Servers**

Stateless servers do not maintain any client information. Every request from a client is accompanied with parameters that are needed for an operation.

For a byte stream file server the operations for a file to be stateless are:

i. Read (filename, position, m, buffer): For this operation the server returns to the client with m bytes of data of the file identified as filename and saves it in the buffer. The value of the bytes is returned to the client and the position for reading is specified as position parameter.

ii. Write (filename, position, m, buffer): This operation takes m bytes of data from the buffer and writes it into the file named filename. The position to start writing in the file is specified by position parameter.



Example of Stateless file server

- In the diagram the file server does not keep track of any file state information resulting from a previous operation. If a client makes two subsequent Read calls then the operation will be Read (filename, 0, 10, buffer), Read (filename, 10, 10, buffer).

In this the client has kept track of file state information.

Stateful servers provide an easier programming paradigm and are more efficient than stateless servers. Stateless servers have a distinct advantage over stateful servers in the event of a failure.

- With stateful servers, if a crashes and restarts later, the state information that it was holding may be lost and the client process might continue its task unaware of the crash.

Stateless servers can be constructed around repeatable operations that make crash recovery easy.

Module 2: Synchronization, mutual exclusion, deadlock

Lecture 1: synchronization algorithms

Synchronization is coordination with respect to time, and refers to the ordering of events and execution of instructions in time. Examples of synchronization include ordering distributed events and ensuring that a process performs an action at a particular time.

Following synchronization-related issues are described in distributed systems:

- Clock synchronization
- Event ordering
- Mutual exclusion
- Deadlock
- Election algorithms

Clock synchronization

It is often important to know when events occurred and in what order they occurred. In a non-distributed system dealing with time is trivial as there is a single shared clock, where all the processes see the same time. On the other hand, in a distributed system, each computer has its own clock. As no clock is perfect, each of these clocks has its own skew which causes clocks on different computers to drift and eventually become out of sync.

- **Physical Clocks**
  - The time difference between two computers is known as drift. Clock drift over time is known as skew. Computer clock manufacturers specify a maximum skew rate in their products.
  - Computer clocks are among the least accurate modern timepieces. „

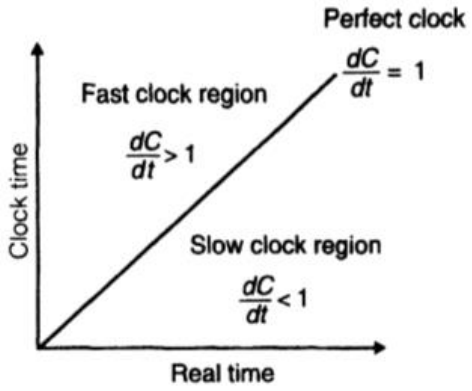
    Inside every computer is a chip surrounding a quartz crystal oscillator to record time. These crystals cost 25 seconds to produce. Average loss of accuracy: 0.86 seconds per day. „

  - This skew is unacceptable for distributed systems. Several methods are now in use to attempt the synchronization of physical clocks in distributed systems:

- We can say that a timer is within specification if there is some constant p such that:

$$1-p<=dC/dT<=1+p$$

- The constant p is the maximum drift rate of the timer.
- On any two given computers, the drift rate will likely differ.
- To solve this problem, clock synchronization algorithms are necessary.



Slow, perfect, and fast clocks.

Clock Synchronization Issues

An important issue in clock synchronization is that time must never run backward because this could cause serious problems, such as the repetition of certain operations that may be hazardous in certain cases. During synchronization a fast clock has to be slowed down. However, if the time of a fast clock is readjusted to the actual time all at once, it may lead to running the time backward for that clock. Therefore, clock synchronization algorithms are normally designed to gradually introduce such a change in the fast running clock instead of readjusting it to the correct time all at once. One way to do this is to make the interrupt routine more intelligent. When an intelligent interrupt routine is instructed by the clock synchronization algorithm to slow down its clock, it readjusts the amount of time to be added to the clock time for each interrupt. For example, suppose that if 8msec is added to the clock time on each interrupt in the normal situation, when slowing down, the interrupt routine only adds 7 msec on each interrupt until the correction has been made. Although not necessary, for smooth readjustment, the intelligent interrupt routine may also advance its clock forward, if it is found to be slow, by adding 9msec on each interrupt, instead of readjusting it to the correct time all at once.

Clock Synchronization Algorithms

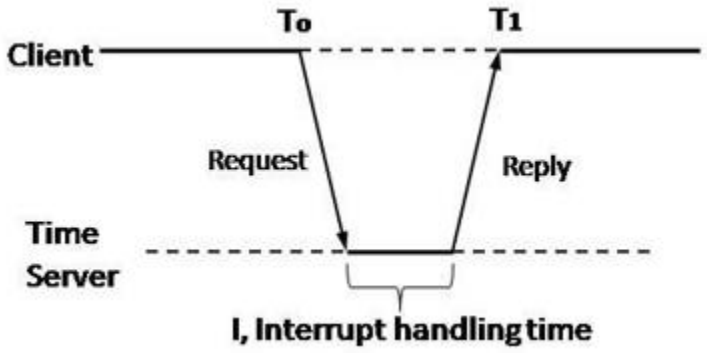
Clock synchronization algorithms may be broadly classified as centralized and distributed.

Centralized clock synchronization algorithms

These have one node with a real-time receiver and are called time server node. The clock time of this node is regarded as correct and used as reference time.

The goal of this algorithm is to keep the clocks of all other nodes synchronized with time server node.

i. **Cristian's Algorithm**



- In this method each node periodically sends a message to the server. When the time server receives the message it responds with a message T, where T is the current time of server node.
- Assume the clock time of client be To when it sends the message and T1 when it receives the message from server. To and T1 are measured using same clock so best estimate of time for propagation is (T1-To)/2.
- When the reply is received at clients node, its clock is readjusted to T+(T1-T0)/2. There can be unpredictable variation in the message propagation time between the nodes hence (T1-T0)/2 is not good to be added to T for calculating current time.
- For this several measurements of T1-To are made and if these measurements exceed some threshold value then they are unreliable and discarded. The average of the remaining measurements is calculated and the minimum value is considered accurate and half of the calculated value is added to T.
- Advantage-It assumes that no additional information is available.
- Disadvantage- It restricts the number of measurements for estimating the value.

ii. The Berkley Algorithm

- This is an active time server approach where the time server periodically broadcasts its clock time and the other nodes receive the message to correct their own clocks.
- In this algorithm the time server periodically sends a message to all the computers in the group of computers. When this message is received each computer sends back its own clock value to the time server. The time server has a prior knowledge of the approximate time required for propagation of a message which is used to readjust the clock values. It then takes a fault tolerant average of clock values of all the computers. The calculated average is the current time to which all clocks should be readjusted.
- The time server readjusts its own clock to this value and instead of sending the current time to other computers it sends the amount of time each computer needs for readjustment. This can be positive or negative value and is calculated based on the knowledge the time server has about the propagation of message.

Distributed clock synchronization algorithms

Distributed algorithms overcome the problems of centralized by internally synchronizing for better accuracy. One of the two approaches can be used:

i. Global Averaging Distributed Algorithms

- In this approach the clock process at each node broadcasts its local clock time in the form of a "resync" message at the beginning of every fixed-length resynchronization interval. This is done when its local time equals To+iR for some integer i, where To is a fixed time agreed by all nodes and R is a system parameter that depends on total nodes in a system.
- After broadcasting the clock value, the clock process of a node waits for time T which is determined by the algorithm.
- During this waiting the clock process collects the resync messages and the clock process records the time when the message is received which estimates the skew after the waiting is done. It then computes a fault-tolerant average of the estimated skew and uses it to correct the clocks.

ii. Localized Averaging Distributes Algorithms

- The global averaging algorithms do not scale as they need a network to support broadcast facility and a lot of message traffic is generated.
- Localized averaging algorithms overcome these drawbacks as the nodes in distributed systems are logically arranged in a pattern or ring.
- Each node exchanges its clock time with its neighbors and then sets its clock time to the average of its own clock time and of its neighbors.

Lecture 2: Event Ordering

Keeping the clocks in a distributed system synchronized to within 5 or 10msec is an expensive and nontrivial task. Lamport [1978] observed that for most applications it is not necessary to keep the clocks in a distributed system synchronized. Rather, it is sufficient to ensure that all events that occur in a distributed system be totally ordered in a manner that is consistent with an observed behavior. For partial ordering of events, Lamport defined a new relation called happened before and introduced the concept of logical clocks for ordering of events based on the happened-before relation. He then gave a distributed algorithm extending his idea of partial ordering to a consistent total ordering of all the events in a distributed system.

Definitions:

Happened Before Relation (->):

This relation captures causal dependencies between events, that is, whether or not events have a cause and effect relation.

This relation (->) is defined as follows:

- a -> b, if a and b are in the same process and a Occurred before b.
- a -> b, if a is the event of sending a message and b is the receipt of that message by another process. If a -> b and b -> c, then a -> c, that is, the relation has the property of transitivity.

Causally Related Events: If event a -> event b, then a casually affects b.

Concurrent Events: Two distinct events a and b are concurrent (a || b) if (not) a -> b and (not) b -> a. That is, the events have no causal relationship. This is equivalent to b || a.

For any two events a and b in a system, only one of the following is true: a -> b, b -> a, or a || b.

Lamport introduced a system of logical clocks in order to make the -> relation possible. It works like this: Each process Pi in the system has its own clock Ci. Ci can be looked at as a function that assigns a number, Ci(a) to an event a. This is the timestamp of the event a in process Pi. These numbers are not in any way related to physical time -- that is why they are called logical clocks. These are generally implemented using counters, which increase each time an event occurs. Generally, an event's timestamp is the value of the clock at the time it occurs.

Conditions Satisfied by the Logical Clock system:

For any events a and b, if a -> b, then C(a) < C(b). This is true if two conditions are met:

- If a occurs before b, then Ci(a) < Ci(b).
- If a is a message sent from Pi and b is the receipt of that same message in Pj, then Ci(a) < Cj(b).

Implementation Rules Required:

Clock Ci is incremented for each event: Ci := Ci + d (d > 0) if a is the event of sending a message from one process to another, then the receiver sets its clock to the max of its current clock and the sender's clock - that is,

$C_j := \max(C_j, t_m + d)$ $(d > 0)$.

Global State

In a fault-tolerant distributed system, backward error recovery requires that the system regularly saves its state onto stable storage. In particular, we need to record a consistent global state, also called a distributed snapshot. In a distributed snapshot, if a process P has recorded the receipt of a message, then there should also be a process Q that has recorded the sending of that message. After all, it must have come from somewhere.

In backward error recovery schemes, each process saves its state from time to time to a locally-available stable storage. To recover after a process or system failure requires that we construct a consistent global state from these local states. In particular, it is best to recover to the most recent distributed snapshot, also referred to as a recovery line. In other words, a recovery line corresponds to the most recent consistent collection of checkpoints, as shown in Fig.

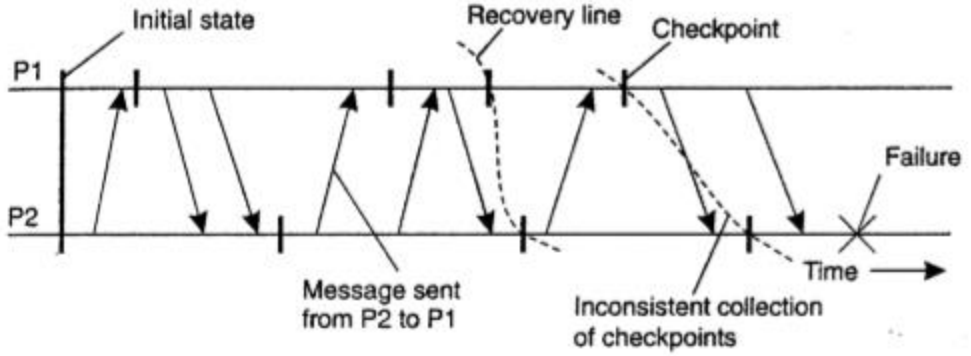


Figure A recovery line.

Lecture 3: Mutual Exclusion

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, a file must not be simultaneously updated by multiple processes. Similarly, use of unit record peripherals such as tape drives or printers must be restricted to a single process at a time. Therefore, exclusive access to such a shared resource by a process must be ensured. This exclusiveness of access is called mutual exclusion between processes. The sections of a program that need exclusive access to shared resources are referred to as critical sections. For mutual exclusion, means are introduced to prevent processes from executing concurrently within their associated critical sections.

Distributed mutual exclusion algorithms can be classified into two different categories. In token-based solutions mutual exclusion is achieved by passing a special message between the processes, known as a token. There is only one token available and whoever has that token is allowed to access the shared resource. When finished, the token is passed on to a next process. If a process having the token is not interested in accessing the resource, it simply passes it on. Token-based solutions have a few important properties.
First, depending on the how the processes are organized; they can fairly easily ensure that every process will get a chance at accessing the resource. In other words, they avoid starvation.

Second, deadlocks by which several processes are waiting for each other to proceed, can easily be avoided, contributing to their simplicity. Unfortunately, the main drawback of token-based solutions is a rather serious one: when the token is lost (e.g., because the process holding it crashed), an intricate distributed procedure needs to be started to ensure that a new token is created, but above all, that it is also the only token.
As an alternative, many distributed mutual exclusion algorithms follow a permission-based approach. In this case, a process wanting to access the resource first requires the permission of other processes. There are many different ways toward granting such permission and in the sections that follow we will consider a few of them.

Centralized Algorithm

„The most simple and straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system: „

- One process is elected as the coordinator. „
- When any process wants to enter a critical section, it sends a request message to the coordinator stating which critical section it wants to access. „
- If no other process is currently in that critical section, the coordinator sends back a reply granting permission. When the reply arrives, the requesting process enters the critical section.
- If another process requests access to the same critical section, it is ignored or blocked until the first process exits the critical section and sends a message to the coordinator stating that it has exited.

The Centralized Algorithm has the following disadvantages:

- The coordinator is a single point of failure.

- If processes are normally ignored when requesting a critical section that is in use, they cannot distinguish between a dead coordinator and "permission denied".
- In a large system, a single coordinator can be a bottleneck.

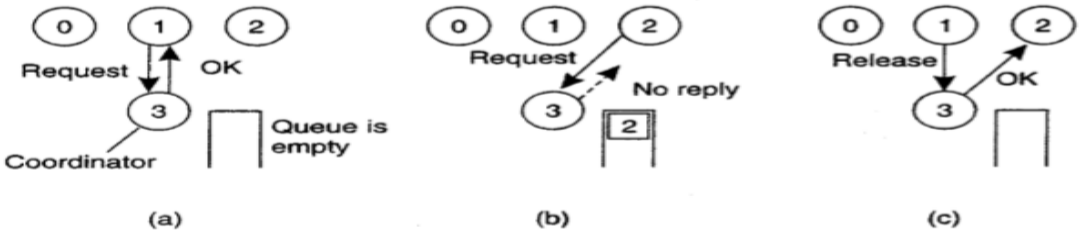


Figure 6.4: (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

Distributed Algorithms

It is often unacceptable to have a single point of failure. Therefore researchers continue to look for distributed mutual exclusion algorithms. The most well-known is by Ricart and Agrawala:

- There must be a total ordering of all events in the system. Lamport's Algorithm can be used for this purpose.
- When a process wants to enter a critical section, it builds a message containing the name of the critical section, its process number, and the current time. It then sends the message to all other processes, as well as to itself.
- When a process receives a request message, the action it takes depends on its state with respect to the critical section named in the message.
  There are three cases:
  - If the receiver is not in the critical section and does not want to enter it, it sends an OK message to the sender.
  - If the receiver is in the critical section, it does not reply. It instead queues the request. „
  - If the receiver also wants to enter the same critical section, it compares the time stamp in the incoming message with the time stamp in the message it has sent out. The lowest time stamp wins. If its own message has a lower time stamp, it does not reply and queues the request from the sending process.

- When a process has received OK messages from all other processes, it enters the critical section. Upon exiting the critical section, it sends OK messages to all processes in its queue and deletes them all from the queue.

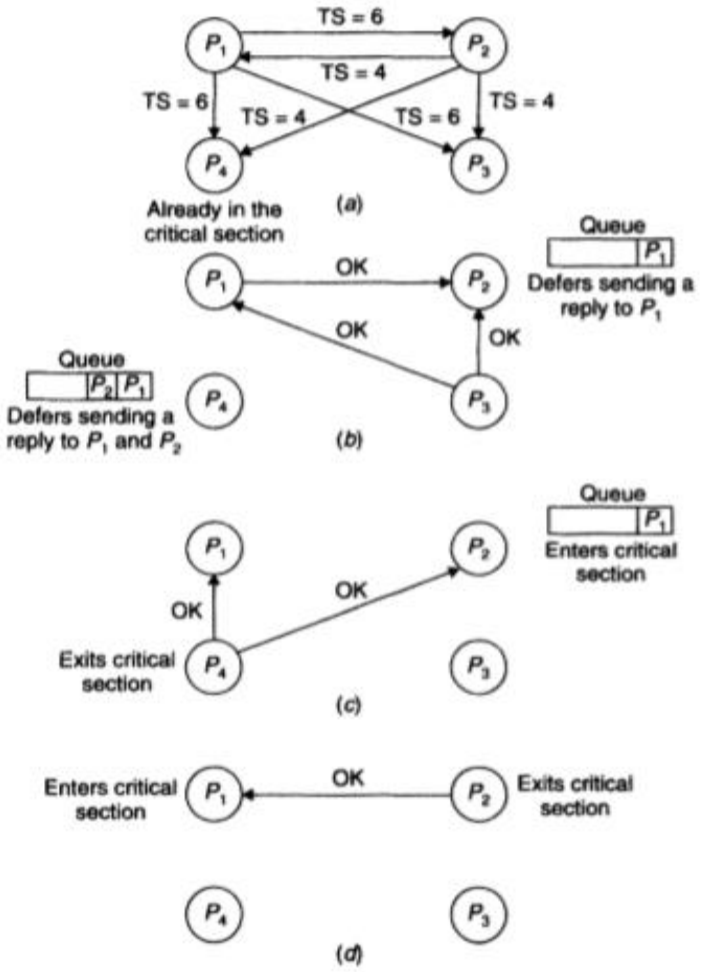


Fig. Example illustrating the distributed algorithm for mutual exclusion: (*a*) status when processes P_1 and P_2 send request messages to other processes while process P_4 is already in the critical section; (*b*) status while process P_4 is still in critical section; (*c*) status after process P_4 exits critical section; (*d*) status after process P_2 exits critical section.

To illustrate how the algorithm works, let us consider the example of Figure a. There are four processes P_I, P_2' P_3' and P_4' While process P_4 is in a critical section, processes P_I and P_2 want to enter the same critical section. To get permission from other processes, processes P_I and P_2 send request messages with timestamps 6 and 4 respectively to other processes (fig. a). Now let us consider the situation in Figure b. Since process P_4 is already in the critical section, it defers sending a reply message to P_I and P_2 and enters them in its queue. Process P_3 is currently not interested in the critical section, so it sends a reply message to both PI and P_2. Process P_2 defers sending a reply message to P_I and enters P_1 in its queue because the timestamp (4) in its own request message is less than the timestamp (6) in P_1's request message. On the other hand, P_I immediately replies to P_2 because the timestamp (6) in its request message is found to be greater than the timestamp (4) of P_2's request message. Next consider the situation in Figure c. When process P_4 exits the critical section, it sends a

reply message to all processes in its queue (in this case to processes P_I and P_2) and deletes them from its queue. Now since process P_2 has received a reply message from all other processes (P_I' P_3, and P_4), it enters the critical section. However, process P_I continues to wait since it has not yet received a reply message from process P2. Finally, when process P2 exits the critical section, it sends a reply message to P_I (Fig. d). Now since process P_I has received a reply message from all other processes, it enters the critical section.

Lecture 4: Token-Passing Approach

In this method, mutual exclusion is achieved by using a single token that is circulated among the processes in the system. A token is a special typeof message that entitles its holder to enter a critical section. For fairness, the processes in the system are logically organized in a ring structure, and the token is circulated from one process to another around the ring always in the same direction (clockwise or anticlockwise).

Suzuki-Kasami's broadcast algorithm

In Suzuki-kasami algorithm, if a site wants to enter the CS and in case if it do not possess the token, it broadcasts a REQUEST message for the token to all other sites. A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of its CS.

Token Consist of:

– Q: Queue of the requesting processes, at most n.
– LN [1...n]: array of integers, LN[j] is the sequence number of the request that Pj executed most recently.

Data Structures:

• REQUEST (j,n): REQUEST message from Pj for its nth request to enter the CS.
• RNi[1..N]: RNi[j] is the largest sequence number in a REQUEST message from Pj received by Pi.
• On receipt of REQUEST (j,n), Pi sets RNi[j] to be max(RNi[j],n).
• If RNi[j] >n, the message is outdated.

This algorithm must efficiently address the following two design issues:

(1) How to distinguish an outdated REQUEST message from a current REQUEST message: Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied. If a site can not determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it .this will not violate the correctness, however, this may seriously degrade the performance.

(2) **How to determine which site has an outstanding request for the CS:** After a site has finished the execution of the CS, it must determine how many sites have an outstanding request for the CS so that the token can be dispatched to one of them.

The first issue is addressed in the following manner: A REQUEST message of site Pj has the form REQUEST (j, n) where n (n=1, 2 ...) is a sequence number which indicates that site Pj is requesting its nth CS execution. A site Pi keeps an array of integers RNi[1..N].where RNi[j] denotes the largest sequence number received in a REQUEST message so far received from site Pj .When site Pi receives a REQUEST(j, n) message, it sets RNi[j]:=max(RNi[j], n).When a site Pi receives a REQUEST(j, n) message, the request is outdated if RNi[j]>n.

The second issue is addressed in the following manner: The token consists of a queue of requesting sites Q, and an array of integers LN [1...N]; where LN[j] is the sequence number of the request which site Pj executed most recently. After executing its CS, a site Pi updates

LN[i]:=RNi[i] to indicate that its request corresponding to sequence number RNi[i] has been executed .At site Pi if RNi[j]=LN[j]+1, then site Pj is currently requesting token.

Algorithm:

Requesting the CS:

– If the requesting site Pi does not have the token, it increments its sequence number RNi[i], and sends a REQUEST (i,sn) message to all other sites.

– When Pj receives the message, it sets RNj[i] to max (RNj[i], sn). If Pj has the idle token, it sends the token to Pi if RNj[i] =LN[i] +1.

Executing the CS: Enter CS when gets token.

Releasing the CS: Having finished the execution of the CS, site Pi takes the following actions:

– Sets LN[i] to Rni[i].

– For every site Pj whose ID is not in the token queue, it appends its ID to the token queue if RNi[j] = LN[j] +1.

– If token queue is nonempty after the above update, it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

Performance

- **Message complexity:** Requires 0 messages if the requesting site holds the idle token. N messages otherwise (N-1 broadcast and 1 to send the token).
- **Synchronization delay:** 0 or T based on if the site holds the token at the time of request.
- **No Starvation:** Token request messages reach all other sites in finite time. Since one of these sites posses the token, the request will be placed to the token Q in finite time. Since there are at most N-1 other requests in front of this request, the request will be granted in finite time.

Lecture 5: Election algorithms

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator (using this as a generic name for the special process). If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special -.Consequently, we will assume that each process has a unique number, for example, its network address (for simplicity, we will assume one process per machine). In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. The algorithms differ in the way they do the location.

Bully algorithm

The process with the highest identity always becomes the coordinator.

When a process P sees that the coordinator is no longer responding to requests it initiates an election by sending ELECTION messages to all processes whose id is higher than its own. If no one responds to the messages then P is the new coordinator. If one of the higher-ups responds, it takes over and P doesn't have to worry anymore.

When a process receives an ELECTION message it sends a response back saying OK. It then holds it's own election (unless it is already holding one). Eventually there is only one process that has not given up and that is the new coordinator. It is also the one with the highest number currently running. When the election is done the new coordinator sends a COORDINATOR message to everyone informing them of the change.

If a process which was down comes back up, it immediately holds an election. If this process had previously been the coordinator it will take this role back from whoever is doing it currently (hence the name of the algorithm).

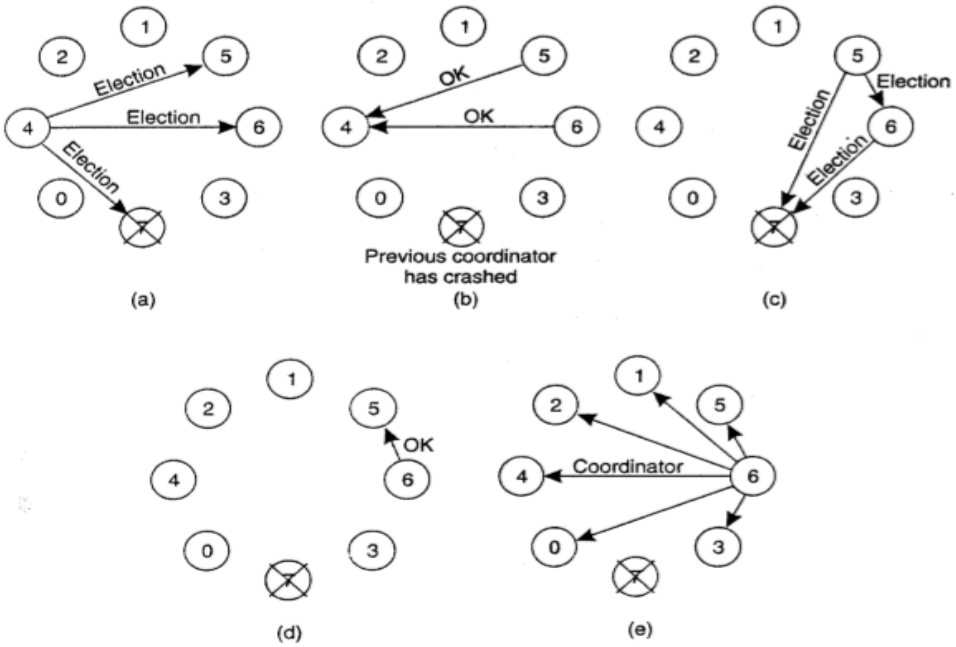


Figure 6. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

Lecture 6: Election algorithms

Ring Algorithm:

- It is based on the use of a ring as the name suggests. But this doesn't use a toke. Processes are physically ordered in such a way that every process knows its successor.
- When any process notices that the coordinator is no longer functioning, it builds up an ELECTION message containing its own number and passes it along the to its successor. If the successor is down, then sender skips that member along the ring to the next working process.
- At each step, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected s the coordinator. At the end, the message gets back to the process that started it.
- That process identifies this event when it receives an incoming message containing its own process number. Then the same message is changed as coordinator and is circulated once again.
- Example: two process, Number 2 and Number 5 discover together that the previous coordinator; Number 7 has crashed. Number 2 and Number 5 will each build an election message and start circulating it along the ring. Both the messages in the end will go to Number 2 and Number 5 and they will convert the message into the coordinator with exactly the same number of members and in the same order. When both such messages have gone around the ring, they both will be discarded and the process of election will re-start.

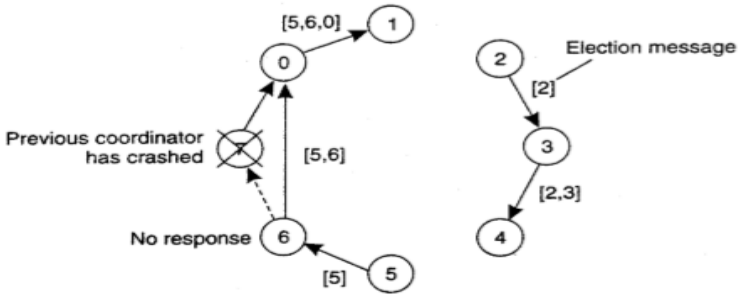


Figure 6. Election algorithm using a ring.

Lecture 7: Distributed Deadlock Detection

- Deadlocks are a fundamental problem in distributed systems.
- A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
- If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur.
- A deadlock is a state where a set of processes request resources that are held by other processes in the set.
- A distributed program is composed of a set of n asynchronous processes p1, p2, . . ., pi, . . . , pn that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.

Necessary Conditions for Deadlock

1. **Mutual-exclusion condition**. If a resource is held by a process, any other process requesting for that resource must wait until the resource has been released.
2**. Hold-and-wait condition**. Processes are allowed to request for new resources without releasing the resources that they are currently holding.
3. **No-preemption condition**. A resource that has been allocated to a process becomes available for allocation to another process only after it has been voluntarily released by the process holding it.
4. **Circular-wait condition**. Two or more processes must form a circular chain in which each process is waiting for a resource that is held by the next member of the chain.

All four conditions must hold simultaneously in a system for a deadlock to occur. If anyone of them is absent, no deadlock can occur. Notice that the four conditions are not completely independent because the circular-wait condition implies the hold-and-wait condition. Although these four conditions are somewhat interrelated, it is quite useful to consider them separately to devise methods for deadlock prevention.

Lecture 8: Control Organization for Distributed Deadlock Detection Algorithms

Algorithms for detecting distributed deadlock can be handled in three different ways:

- ✓ Centralized
- ✓ Distributed
- ✓ Hierarchical

Assume that the network supports reliable communication.

Centralized:

One central site sets up a global WFG and searches for cycles. All decisions are made by the central control node.

- It must maintain the global WFG constantly or
- Periodically reconstruct it.

The main advantage is that this permits the use of relatively simple algorithms.
The disadvantages include the following:

- There is one, single point of failure.
- There can be a communication bottleneck around the site due to all the WFG information messages.
- Furthermore, this traffic is independent of the formation of any deadlock.

Distributed:

In a distributed control organization,

- All sites have an equal amount of information.
- All sites make decisions based on local information.
- All sites bear equal responsibility for the final decision in detecting deadlock.
- All sites expend equal effort to the final decision.
- The global WFG is spread across the sites.
- Deadlock detection is initiated whenever a process thinks there might be a problem.
- Several sites can initiate the detection at the same time.

The advantages include the following:

There is no central point of failure.
A single node failure cannot cause a crash.
There is no *one* site with heavy traffic due to the detection algorithm.
The algorithm is only initiated when process(es) feel there might be a problem.
The algorithm is not run periodically, only when needed.

The main disadvantage is that resolution may be difficult, as not all sites may be aware of the processes involved in the deadlock.
The proof of correctness for this type of algorithm may be difficult.

Hierarchical:

The sites (nodes) are logically connected in a hierarchical structure (such as a tree).
A site can detect deadlock in its descendants.
This type of algorithm has the best of both the centralized and the distributed deadlock detection algorithms.
For efficiency purposes, it is best to keep clusters of interacting processes together in the hierarchy.

Lecture 9: Deadlock Handling Strategies

- There are three strategies for handling deadlocks, viz., deadlock prevention, deadlock avoidance, and deadlock detection.
- Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay.
- Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting
- a process which holds the needed resource.
- This approach is highly inefficient and impractical in distributed systems.
- In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system).
- However, due to several problems, deadlock avoidance is impractical in distributed systems.
- Deadlock detection requires examination of the status of process-resource interactions for presence of cyclic wait.
- Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.
- Deadlock handling using the approach of deadlock detection entails addressing two basic issues: First, detection of existing deadlocks and second resolution of detected deadlocks.
- Detection of deadlocks involves addressing two issues: Maintenance of the WFG and searching of the WFG for the presence of cycles (or knots).

Lecture 10: Centralized and Distributed deadlock detection algorithms

Distributed deadlock detection algorithms can be divided into four classes:

- ✓ path-pushing
- ✓ edge-chasing
- ✓ diffusion computation
- ✓ global state detection.

Path-Pushing Algorithms

- In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG.
- The basic idea is to build a global WFG for each site of the distributed system.
- In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighboring sites.
- After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.
- This feature of sending around the paths of global WFG has led to the term path-pushing algorithms.

Edge-Chasing Algorithms

- In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is be verified by propagating special messages called probes, along the edges of the graph.
- These probe messages are different than the request and reply messages.
- The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.
- Whenever a process that is executing receives a probe message, it discards this message and continues.
- Only blocked processes propagate probe messages along their outgoing edges.
- Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short.

Module 3: Distributed file systems

Lecture 1: Issues in the design of distributed file systems

In a computer system, a file is a named object that comes into existence by explicit creation, is immune to temporary failure in the system and persists until explicitly destroyed. Two main purposes of using files:

1. Permanent storage of information on a secondary storage media – this is achieved by storing a file on a secondary storage media such as a magnetic disk.

2. Sharing of information between applications -

A file system is a subsystem of an operating system that performs file management activities such as organization storing retrieval, naming, sharing, and protection of files.

A distributed file system provides the following types of services:

1. Storage service

Allocation and management of space on a secondary storage device thus providing a logical view of the storage system.

2. True file service

Includes file-sharing semantics, file-caching mechanism, file replication mechanism, concurrency control, multiple copy update protocol etc.

3. Name service

Responsible for directory related activities such as creation and deletion of directories, adding a new file to a directory, deleting a file from a directory, changing the name of a file, moving a file from one directory to another etc.

Desirable features of a distributed file system:

1. Transparency

    - Structure transparency

      Clients should not know the number or locations of file servers and the storage devices. Note: multiple file servers provided for performance, scalability, and reliability.

    - Access transparency

      Both local and remote files should be accessible in the same way. The file system should automatically locate an accessed file and transport it to the client's site.

    - Naming transparency

      The name of the file should give no hint as to the location of the file. The name of the file must not be changed when moving from one node to another.

    - Replication transparency

      If a file is replicated on multiple nodes, both the existence of multiple copies and their locations should be hidden from the clients.

    - Failure transparency

      The client and client programs should operate correctly after a server failure.

    - Location Transparency

      A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.

    - Migration transparency

      Files should be able to move around without the client's knowledge.

2. User mobility

Automatically bring the user's environment (e.g. user's home directory) to the node where the user logs in.
Furthermore,theperformancecharacteristicsthefilesystemshouldnotdiscourageusersfromaccessingtheirfilesfromworkstationsotherthantheoneatwhichtheyusuallywork.

3. Performance

Performance is measured as the average amount of time needed to satisfy client requests. This time includes CPU time + time for accessing secondary storage + network access time. It is desirable that the performance of a distributed file system be comparable to that of a centralized file system.

4. Simplicity and ease of use

User interface to the file system be simple and number of commands should be as small as possible.

5. Scalability

The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).

6. High availability

A distributed file system should continue to function in the face of partial failures such as a link failure, a node failure, or a storage device crash.

A highly reliable and scalable distributed file system should have multiple and independent file servers controlling multiple and independent storage devices.

7. High reliability

Probability of loss of stored data should be minimized. System should automatically generate backup copies of critical files.

8. Data integrity

Concurrent access requests from multiple users who are competing to access the file must be properly synchronized by the use of some form of concurrency control mechanism. Atomic transactions can also be provided.

9. Security

Users should be confident of the privacy of their data.

10.Heterogeneity

File service should be provided across different hardware and operating system platforms.

Lecture 2:File Models

Different file systems use different conceptual models of a file. The two most commonly used criteria for file modeling are structure and modifiability. File models based on these criteria are described below:

1. Unstructured and Structured Files

   In the unstructured model, a file is an unstructured sequence of bytes. The interpretation of the meaning and structure of the data stored in the files is up to the application (e.g. UNIX and MS-DOS). Most modern operating systems use the unstructured file model.

   In structured files (rarely used now) a file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different sizes.

2. Mutable and Immutable Files

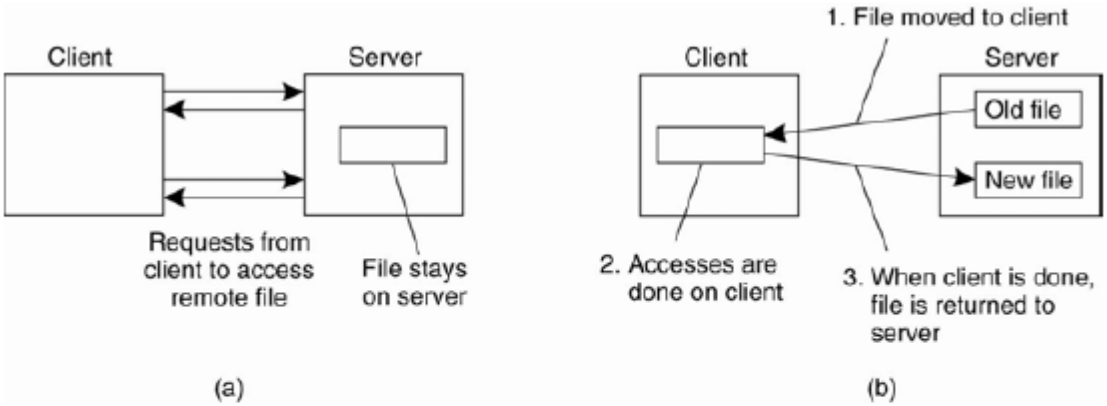
   Based on the modifiability criteria, files are of two types, mutable and immutable. Most existing operating systems use the mutable file model. An update performed on a file overwrites its old contents to produce the new contents.

   Immutable files cannot be altered after they have been closed. In order to change a file, instead of overwriting the contents of the existing file a new file must be created. This file may then replace the old one as a whole. This approach to modifying files does require that directories (unlike files) be updatable. Problems with this approach include a race condition when two clients try to replace the same file as well as the question of what to do with processes that are reading a file at the same time as it is being replaced by another process.

File accessing Models:

The manner in which a client's request to access a file is serviced depends on the file accessing model used by the file system. The file accessing model of a distributed file system mainly depends on two factors the method used for accessing remote files and the unit of data access.

To provide a remote system with file service, we will have to select one of two models of operation. One of these is the **upload/download model (Data caching model)**. In this model, there are two fundamental operations: *read file* transfers an entire file from the server to the requesting client, and *write file* copies the file back to the server. It is a simple model and efficient in that it provides local access to the file when it is being used. Three problems are evident. It can be wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file
The second model is **a remote access model**. The file service provides remote operations such as *open*, *close*, *read bytes*, *write bytes*, *get attributes*, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it.



- Remote access model
  - Work done at the server
  - Consistent sharing (+)
  - Server may be a bottleneck (-)
  - High communication cost (-)
- Upload/download model
  - Work done at the client
  - Low communication cost (+)
  - Consistency is harder to maintain (-)

Unit of data transfer

It refers to fraction of file data that is transferred to and from client as a result of single read write operation. The Four data transfer models are:-

File level transfer model, Block level transfer model, Byte level transfer model, Record level transfer model

*File level transfer model:*When the operation requires file data to be transferred across the network in either direction between client and server, the whole file is moved.
The advantages are:

1. Efficient as network protocol overhead is required only once.
2. Better scalability becauseas it requires fewer access to file server and reduce server load and network traffic.
3. Disk access routines on server can be better optimized.
4. Offers degree of resiliency to server and network failure.
   The disadvantage is it requires sufficient storage space. Example are amoeba, CFS, Andrew file system

Block level transfer model: File data transfer take place in units of data blocks. A file block is contiguous portion of file and fixed in length.Advantage- it does not required large storage space.It also eliminates the need to copy the entire file when only a small portion of the file data is needed.
Disadvantage –it has more network traffic and more network protocol overhead. Therefore, this model has poor performance as compared to the file level transfer model. Example are Sun microsystem's NFS, Apollo domain file system.

Byte level transfer model: In this model, file data transfer take place across the network in either direction between client and server take place in units of bytes. It provides maximum flexibility but difficulty in cache management.

Record level transfer model:The three file data model discussed above are commonly suitable with unstructured file models. The record level transfer model is suitable with structured files. Here, file data transfer take place in unit of records. Example RSS(research storage system).

Lecture 3: File caching schemes

File caching has been implemented in several file systems for centralized time sharing systems to improve file I/O performance (e.g.,UNIX). The idea in file caching in these systems is to retain recently accessed file data in main memory, so that repeated accesses to the same information can be handled without additional disk transfers. Because of locality in file access patterns, file caching reduces disk transfers substantially, resulting in better overall performance of the file system.

Inimplementingafile-cachingschemeforacentralizedfilesystem,onehastomakeseveralkeydecisions,suchasthegranularityofcacheddata,cachesize,andthereplacement policy.Inadditiontotheseissues,afilecachingschemeforadistributedfilesystemshouldalsoaddressthefollowingkeydecisions:

1. Cache location

   Cachelocationreferstotheplacewherethecacheddataisstored.Assumingthattheoriginallocationofafileisonitsserver'sdisk,therearethreepossiblecachelocationsinadistributedfilesystem.
   a. Server's main memory
   b. Client's disk
   c. Client's main memory

Cachelocation	Access cost on cache hit	Advantages
Server's main memory	One network access	• Easy to implement Totally transparent to the clients • Easy to keep the original file and cached • EasytosupportUNIX-likefile-sharingsemantics
Server's main memory	One disk access	• Reliabilityagainstcrashes • largestoragecapacity • Suitable forsupportingdisconnectedoperation • Contributestoscalabilityandreliability
Client's main memory	-	• Maximum Performancegain • Permits workstations to be diskless • Contributestoscalabilityandreliability

2. Modificationpropagation

Adistributedfilesystemmayuseoneofthemodificationpropagationschemesdescribedbelow.The filesharingsemanticssupportedbythedistributedfilesystemdependsgreatlyonthemodificationpropagationschemeused.Furthermore themodificationpropagationschemeusedhasacriticaleffectonthesystemsperformanceandreliability.

Write-through Scheme:

Inthisscheme,whenacacheentryismodified,thenewvalueisimmediatelysenttotheserverforupdatingthemastercopyofthefile.Thisschemehastwomain advantages – high degree ofreliabilityandsuitabilityforUNIX-likesemantics.Sinceeverymodificationis immediatelypropagatedtotheserverhavingthemastercopyofthefile,theriskofupdateddatagetting lost
whenaclientcrashes)isverylow.Amajordrawbackofthisschemeisitspoorwriteperformance.This isbecauseeachwriteaccesshastowaituntiltheinformationiswrittentothemastercopyof theserver.Noticethatwiththewrite-throughschemetheadvantagesofdatacachingareonlyforreadaccessesbecausetheremoteservice methodisbasicallyusedforallwriteaccesses.Therefore,thisschemeissuitableforuseonlyinthosecasesinwhichtheratioofread-to-writeaccessesisfairlylarge.

Delayed-WriteScheme:

Althoughthewrite-throughschemehelpsonreads,itdoesnothelpinreducingthenetworktrafficforwrites.Therefore,toreducenetworktrafficforwritesaswell,somesystemsusethedelayed-writescheme.Inthisscheme,whenacacheentryismodified,thenewvalueiswrittenonlytothecacheandtheclientjustmakesanotethatthecacheentryhasbeenupdated.Sometimelater,allupdatedcacheentriescorrespondingtoafileareg atheredtogetherandsenttotheserveratatime.Dependingonwhenthemodificationsaresenttothefileserver,delayed-writepoliciesareofdifferenttypes.Threecommonlyusedapproachesareasfollows:

a. *<u>Writeonejectionfromcache</u>*

Inthismethod,modifieddatainacacheentryissenttotheserverwhenthecachereplacement

policyhasdecidedtoejectitfromthe clientscache.Thismethodcanresultingoodperformance,butsomedatacanresideintheclient scacheforalongtimebeforetheyaresenttotheserver.Suchdataaresubjecttoreliabilityproblem.

b. *Periodicwrite*

Inthismethod,thecacheisscannedperiodically,atregularintervals,andanycacheddatathathavebeenmodifiedsincethelastscanaresenttotheserver.

c. Writeonclose

Inthismethod,themodificationsmadetoacacheddatabyaclientaresenttotheserverwhenthecorrespondingfileisclosedbytheclient.Noticethatthewrite-on-closepolicyisaperfectmatchforthesessionsemantics.However,itdoesnothelpmuchinreducingnetworktrafficforthosefilesthatareopenforveryshortperiodsorarerarelymodified.Furthermore,thecloseoperationtakesalongtimebecauseallmodifieddatamustbewrittentotheserverbeforetheoperationcompletes.Therefore,thispolicyshouldbeusedonlyincasesinwhichfilesareopenforlongperiodsandarefrequentlymodified.

3. Cachevalidation

A file data may simultaneously reside in the cache of multiple nodes. The modification propagation policy only specifies when the master copy of a file at a server node is updated upon modification of a cache entry.

It does not tell anything about when the file data residing in the cache of other nodes was updated. As soon as other nodes get updated, the client's data become outdated or stale. Thus the consistency of the clients' cache has to be checked and must be consistent with the master copy of the data. Validation is done in two ways:

a. Client initiated approach:

Here client checks for new updates before it accesses its data or it goes with the periodic checking mechanism i.e. client checks for updates after regular intervals of time. Here the pull mechanism is implemented where the client pulls for updates.

b. Server initiated approach:

Here the server is responsible for sending periodic updates to all its clients. The Push protocol is user where she server pushes the new updates to all its clients.

Lecture 4: Fault Tolerance

Fault tolerance is an important issue in the design of a distributed file system. Various types of faults could harm the integrity of the data stored by such a system. For instance, aprocessor loses the comments of its main memory in the event of a crash. Such a failure could result in logically complete but physically incomplete file operations, making the data that are stored by the file system inconsistent. Similarly, during a request processing, the server or client machine may crash, resulting in the loss of state information of the file being accessed. This may have an uncertain effect on the integrity of file data. Also, other adverse environmental phenomena such as transient faults (caused by electromagnetic fluctuations) or decay of disk storage devices may result in the loss or corruption of data stored by a file system. A portion of a disk storage device is said to be 'decay'. The data on that portion of the device are irretrievable. The primary file properties that directly influence ability of a distributed file system to tolerate faults are as follows.

1. **Availability:** Availability of a file refers to the fraction of time for which the file is available for use. Note that the availability property depends on the location of the file and the locations of its clients (users). For example, if a network is partitioned due to a communication link failure, a file may be available to the clients of some nodes, but at the same time, it may not be available to the clients of other nodes. Replication is a primary mechanism for improving the availability of a file.

2. **Robustness:** Robustness of a file refers to its power to survive crashes of the storage device and decays of the storage medium on which it is stored. Storage devices that are implemented by using redundancy techniques, such as stable storage device, are often used to store robust files. Note that a robust file may not be available until the faulty component has been recovered. Furthermore, unlike availability, robustness is independent of either the location of the file or the location of its clients.

On the other hand, if a failure occurs that causes a sub transaction to abort before its completion, all of its tentative updates are undone, and its parent is notified. The parent maythen choose to continue processing and try to complete its task using an alternative method or it may abort itself. Therefore, the abort of a sub transaction may not necessarily cause its ancestors to abort. However, if a failure causes an ancestor transaction to abort, the updates of all its descendant transactions (That have already committed) have to be undone. Thus no updates performed within an entire transaction family are made permanent until the top level transaction commits. Only after the top level transaction commits is success reported to the client.

Stable Storage

In context of crash resistance capability, storage may be broadly classified into three types:

a. Volatile storage, such as RAM, which can not withstand power failures or machine crashes. That is, the data stored in a volatile storage is lost in the event of a power failure or a machine crash.
b. Nonvolatile storage such as a disk, which can withstand CPU failures but cannot withstand transient I/O faults and decay of the storage media. Although fairly reliable, non volatile storage media such as a disk have complicated failure modes and may prove to be insufficiently reliable for storing critical data.
c. Stable Storage, which can even withstand transient faults and decay of the storage media. It is a storage approach introduced by Lampson[1981].

   The basic idea of stable storage is to use duplicate storage devices to implement a stable device and to try to ensure that any period when only one of the two component devices is operational is significantly less than the mean time between failures MTBF of a stable device.

Effect of Service Paradigm on fault tolerance

Aservermaybeimplementedbyusinganyoneofthefollowingtwoserviceparadigms-statefulorstateless.Thetwoparadigmsaredistinguishedbyoneaspectoftheclientserver relationship – whetherornotthehistoryoftheservicedrequestsbetweenaclientandaserveraffectstheexecutionofthenextservicerequest.Thestatefulapproachdependsonthehistoryoftheservicedrequests butthestatelessapproachdoesnotdependonit.

StatefulFileServers

Astatefulfileservermaintainsclient's stateinformationfromoneaccessrequesttothenext.Thatisfortwosubsequentrequestsmadebyaclienttoastatefulserverforaccessingafile somestateinformationpertainingtotheserviceperformedfortheclientasaresultofthefirstrequestexecutionisstoredbytheserverprocess.Thisstateinformationissubsequentlyusedwhenexecutingthesecondrequest.

To illustrate how astateful file server works letusconsiderafileserverforbyte-streamfilesthatallowsthefollowingoperationsonfiles:

Open (filename, mode): This operation is used to open a fileidentifiedbyfilenameinthespecifiedmode.Whenafileisopened itsread-writepointerissettozeroandtheserverreturnstotheclientafileidentifierfid thatisusedbytheclientforsubsequentaccessestothatfile.

Read(fid, n, buffer): Thisoperationisusedtogetnbytesofdatafromthefileidentifiedbyfid

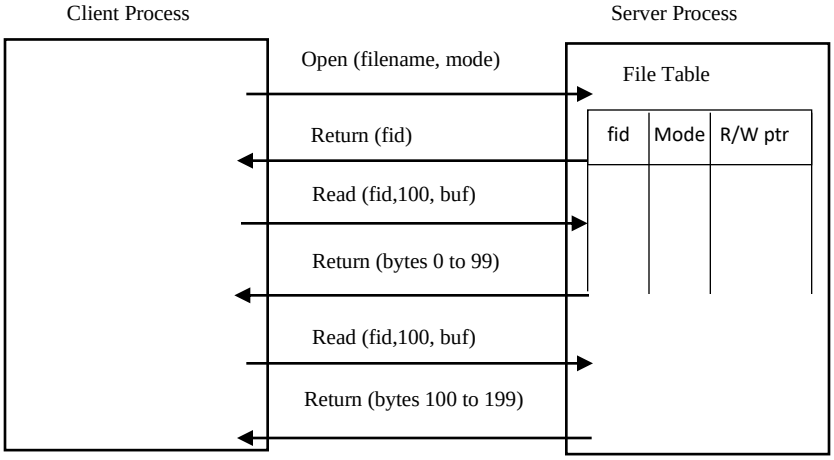
intothespecifiedbuffer.Whentheserverexecutesthisoperation,itreturnstotheclientnbytesoffiledatastartingfromthebytecurrentlyaddressedbytheread-writepointerandthenincrementstheread-writepointerbyn.

Write (fid,n,buffer): Onexecutionofthisoperation,theservertakesnbytesofdatafromthespecifiedbuffer,writesitintothefileidentifiedbyfidatthebytepositioncurrentlyaddressedbytheread-writepointer,andthenincrementstheread-writepointerbyn.

Seek (fid, position): Thisoperationcausestheservertochangethevalueofthereadwritepointerofthefileidentified by fidtothenewvaluespecifiedasposition.

Close (fid): Thisstatementcausestheservertodeletefrom its file-tablethefilestateinformationofthefileidentifiedby fid.

Thefileservermentionedaboveisstatefulbecauseitmaintainsthecurrentstateinformationforafilethathasbeenopenedforusebyaclient.Therefore,asshowninFigure,afteropeningafile,ifaclientmakestwosubsequentRead (fid,100,buf) requests,forthefirstrequestthefirst100bytes (bytes0to99) willbereadandforthesecondrequestthenext100bytes (bytes100to199) willberead.



An example of stateful file server

Stateless File Servers

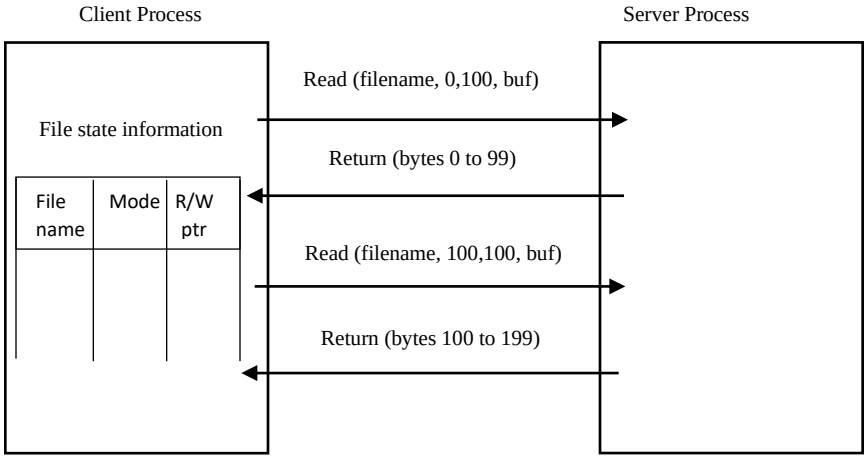
Astatelessfileserverdoesnotmaintainanyclientstateinformation.Thereforeeveryrequestfromaclientmustbeaccompaniedwithallthenecessaryparameterstosuccessfullycarryoutthedesiredoperation.

Forexample,aserverforbyte-streamfilesthatallowsthefollowingoperationsonfilesisstateless:

Read (filename,position,n,buffer): Onexecutionofthisoperation theserverreturnstotheclientnbytesofdataofthefileidentifiedbyfilename.

Thereturneddataisplacedinthespecifiedbuffer.
Thevalueoftheactualnumberofbyteswrittenisalsoreturnedtotheclient.Thepositionwithinthefilefrom wheretobeginreadingisspecifiedasthepositionparameter.

Write (filename,position, n,buffer):Whentheserverexecutesthisoperation ittakesnbytesofdatafromthespecifiedbufferandwritesitintothefileidentifiedbyfilename. Thepositio nparameterspecifiesthebytepositionwithinthefilefromwheretostartwriting.Theserverreturnstothecl ienttheactualnumberofbyteswritten.



An example of stateless file server

Lecture 5: Examples of Sun NFS,Andrew filestore

Network File System (NFS)

NFS is a remote access DFS that was introduced by Sun in 1985. The currently used version isversion 3, however a new version (4) has also been defined. NFS integrates well into Unix's modelof mount points, but does not implement Unix semantics. NFS servers are stateless (i.e., NFSdoes not provide open & close operations). It supports caching, but no replication.NFS has been ported to many platforms and, because the NFS protocol is independent of theunderlying file system, supports many different underlying file systems. On Unix, an NFS serverruns as a daemon and reads the file /etc/export to determine what directories are exported towhom under which policy (for example, who is allowed to mount them, who is allowed to accessthem, etc.). Server-side caching makes use of file caching as provided by the underlying operatingsystem and is, therefore, transparent.On the client side, exported file systems can be explicitly mounted or mounted on demand(called automounting). NFS can be used on diskless workstations so does not require local diskspace for caching files. It does, however, support client-side caching, and allows both file contentsas well as file attributes to be cached. Although NFS allows caching, it leaves the specifics up tothe implementation. As such, file caching details are implementation specific. Cache entries aregenerally discarded after a fixed period of time and a form of delayed write-through is employed.Traditionally, NFS trusts clients and servers and thus has only minimal security mechanismsin place. Typically, the clients simply pass Unixuser ID and group ID of the process performinga request to the server. This implies that NFS users must not have root access on the clients,
otherwise they could simply switch their identity to that of another user and then access thatuser's files. New security mechanisms have been put in place, but they also have their drawbacks:Secure NFS using Diffie-Hellman public key cryptography is more complex to implement and tomanage the keys, and the key lengths used are too short to provide security in practice. UsingKerberos would turn NFS more secure, but it has high entry costs.

Andrew File System (AFS)

The Andrew File System (AFS) is a DFS that came out of the Andrew research project at CarnegieMellon University (CMU). Its goal was to develop a DFS that would scale to all computerson the university's campus. It was further developed into a commercial product and an open-source branch was later released under the name "OpenAFS". AFS offers the same API as Unix,implements Unix semantics for processes on the same machine, but implements write-on-closesemantics globally. All data in AFS is mounted in the /afs directory and organised in cells (e.g./afs/cs.cmu.edu). Cells are administrative units that manage users and servers.Files and

directories are stored on a collection of trusted servers called Vice. Client processesaccessing AFS are redirected by the file system layer to a local user-level process called Venus (theAFS daemon), which then connects to the servers. The servers serve whole files, which are cachedas a whole on the clients' local disks. For cached files a callback is installed on the correspondingserver. After a process finishes modifying a file by closing it, the changes are written back to theserver. The server then uses the callbacks to invalidate the file in other clients' caches. As a result,clients do not have to validate cached files on access (except after a reboot) and hence there isonly very little cache validation traffic. Data is stored on flexible volumes, which can be resizedand moved between the servers of a cell. Volumes can be marked as read only, e.g. for softwareinstallations.AFS does not trust Unix user IDs and instead uses its own IDs which are managed at acell level. Users have to authenticate with Kerberos by using the klog command. On successfulauthentication, a token will be installed in the client's cache managers. When a process tries toaccess a file, the cache manager checks if there is a valid token and enforces the access rights.Tokens have a time stamp and expire, so users have to renew their token from time to time. Authorization is implemented by directory-based ACLs, which allow finer grained access rightsthan Unix.

Lecture 6: CODA file system, OSF DCE

Coda

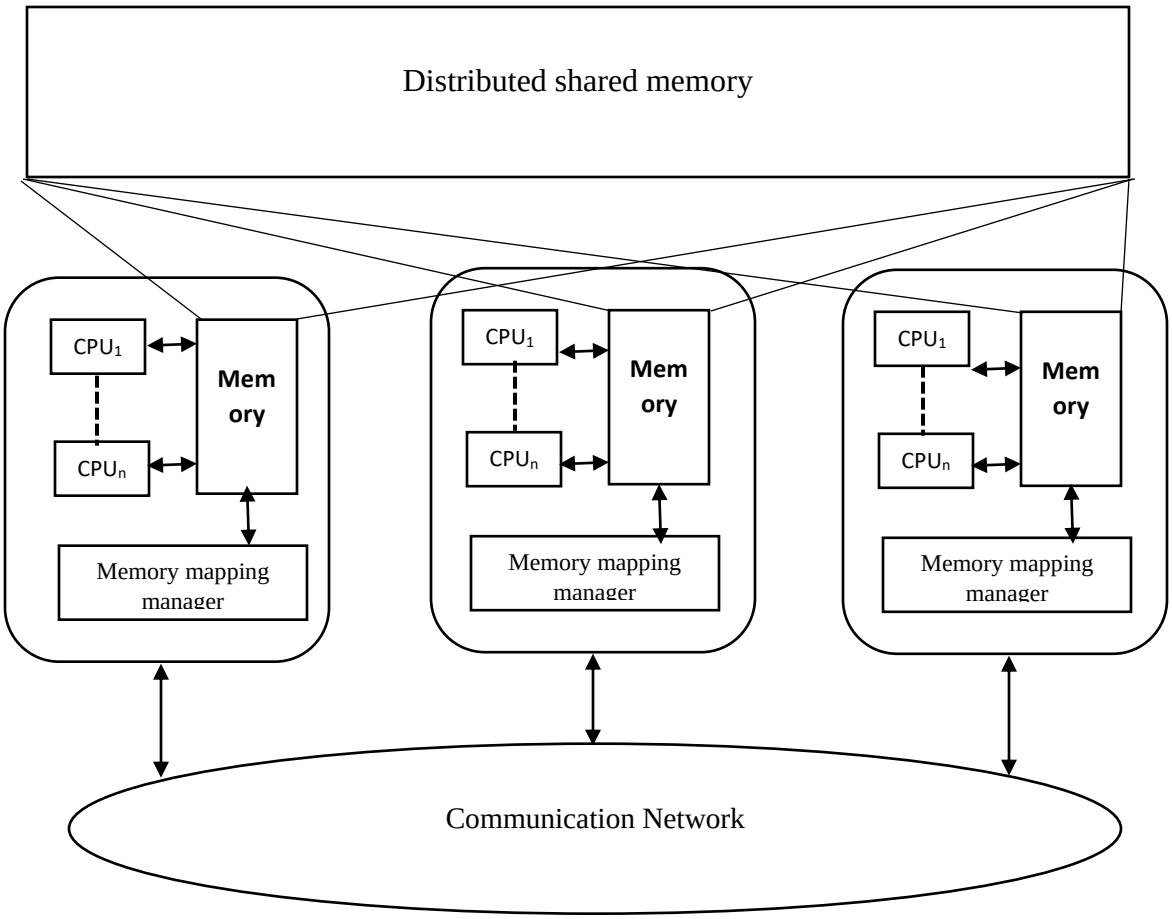
Coda is an experimental DFS developed at CMU by M. Satyanarayanan's group, it is the successor
of the Andrew File System (AFS) but supports disconnected, mobile operation of clients. Its design
is much more ambitious than that of NFS.Coda has quite a number of similarities with AFS. On the client side, there is only a singlemount point /coda. This means that the name space appears the same to all clients (and filestherefore have the same name at all clients). File names are location transparent (servers cannotbe distinguished). Coda provides client-side caching of whole files. The caching is implementedin a user-level cache process called Venus. Coda provides Unix semantics for files shared byprocesses on one machine, but applies write-on-close (session) semantics globally. Because highavailability is one of Coda's goals access to a cached copy of a file is only denied if it is known tobe inconsistent.In contrast to AFS, Coda supports disconnected operation, which works as follows. Whiledisconnected (a client is disconnected with regards to a file if it cannot contact any servers thatserve copies of that file) all updates are logged in a client modification log (CML). Upon reconnection, the operations registered in the CML are replayed on the server. In order to allow clients towork in disconnected mode, Coda tries to make sure that a client always has up-to-date cachedcopies of files that they might require. This process is called file hoarding. The system builds auser hoard database which it uses to update frequently used files using a process called a hoardwalk. Conflicts upon reconnection are resolved automatically where possible, otherwise, manualintervention becomes necessary.

Files in Coda are organised in organisational units called volumes. A volume is a small logicalunit of files (e.g., the home directory of a user or the source tree of a program). Volumes can bemounted anywhere below the /coda mount point (in particular, within other volumes).Coda allows files to be replicated on read/write servers. Replication is organised on a pervolume basis, that is, the unit of replication is the volume. Updates are sent to all replicassimultaneously using multicast RPCs (Coda defines its own RPC protocol that includes a multicastRPC protocol). read operations can be performed at any replica.

Lecture 7:Architecture and motivations.

Distributed shared memory (DSM) system is a resource management component of a distributed operating system that implements the shared memory in distributed systems, which have no physically shared memory. The shared memory model provides a virtual address space that is shared among allnodes in a distributed system.

- In systems that support DSM, data moves between secondary memory and mainmemory as well as between main memories of different nodes.
- Each node can own data stored in the shared address space, the ownership can changewhen data moves from one node to another.



Distributed shared memory (DSM)

- When a process accesses data in the shared address space, a mapping manager mapsthe shared memory address to the physical memory.

- To reduce delays due to communication latency, DSM may move data at the shared memory address from a remote node to the node that is accessing data.
- The OS gets the page from another processor over the network.
- DSM systems hide this explicit data movement and provide a simpler abstraction for sharing data that programmers are already well versed with.
- DSM systems allow complex structures to be passed by reference, thus simplifying the development of algorithms for distributed applications.
- DSM takes advantage of the locality of reference exhibited by programs and thereby cuts down on the overhead of communicating over the network.
- DSM systems are cheaper to build than tightly coupled multiprocessor systems.
- Large memory can be used to efficiently run programs that require large memory without incurring disk latency due to swapping in traditional distributed systems.
- DSM systems do not suffer from this drawback and can easily be scaled upwards.
- Programs written for shared memory multiprocessors can in principle be run on DSM systems without any changes.

Lecture 8: Algorithms for implementing DSM.

Important issues for designing the algorithms are

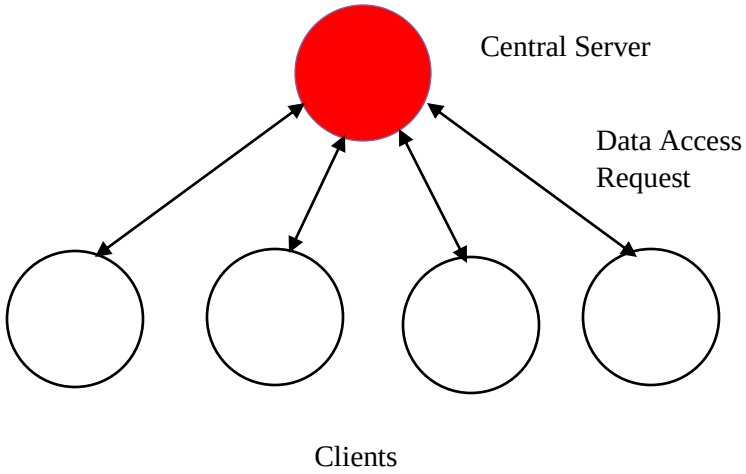
- Granularity
- Structure of shared-memory space
- Memory coherence and access synchronization
- Data location and access
- Replacement strategy
- Thrashing
- Heterogeneity

Central issues in the implementation of DSM:

- How to keep track of the location of remote data.
- How to overcome the communication delays and high overhead associated with the execution of communication protocols in distributed systems when accessing remote data.
- How to make shared data concurrency accessible at several nodes in order to improve system performance.

Algorithms for implementing DSM

1. **The Central-Server Algorithm**



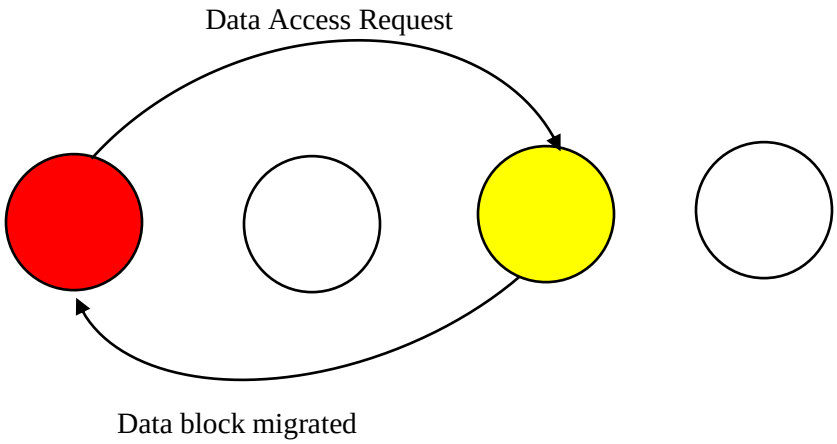
Client	**Central Server**
Send data request	
	Receive Request Perform data access Send response
Receive response	

- Central server maintains all shared data
    - Read request: returns data item
    - Write request: updates data and returns acknowledgement message
- Implementation
    - A timeout is used to resend a request if acknowledgment fails
    - Associated sequence numbers can be used to detect duplicate write requests
    - If an application's request to access shared data fails repeatedly, a failure condition is sent to the application
- Issues: performance and reliability
- Possible solutions
    - Partition shared data between several servers
    - Use a mapping function to distribute/locate data

2. **The Migration Algorithm**

- Operation
    - Ship (migrate) entire data object (page, block) containing data item to requesting location
    - Allow only one node to access a shared data at a time
- Advantages
    - Takes advantage of the locality of reference
    - DSM can be integrated with VM at each node
        - Make DSM page multiple of VM page size
        - A locally held shared memory can be mapped into the VM page address space
        - If page not local, fault-handler migrates page and removes it from address space at remote node

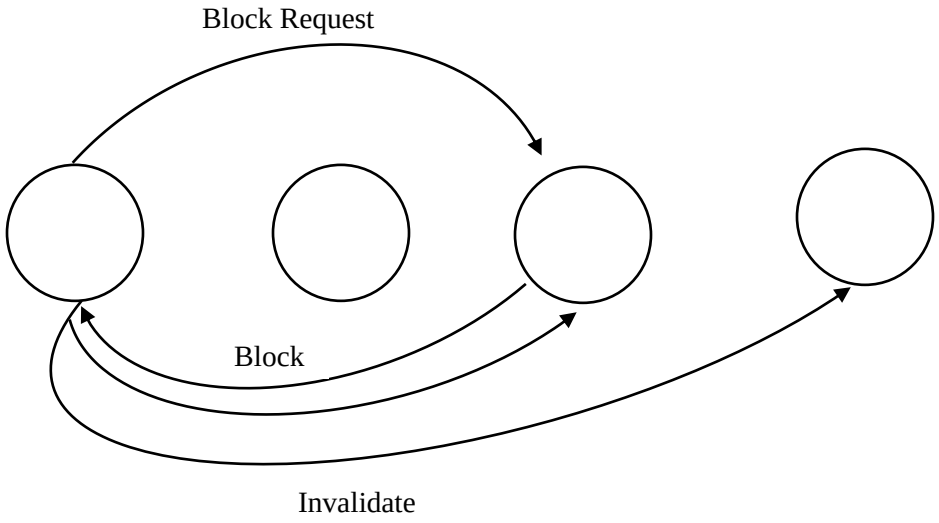
- To locate a remote data object:
    - Use a location server
    - Maintain hints at each node
    - Broadcast query
- Issues
    - Only one node can access a data object at a time
    - Thrashing can occur: to minimize it, set minimum time data object resides at a node



3. **The Read-Replication Algorithm**
    - Replicates data objects to multiple nodes
    - DSM keeps track of location of data objects
    - Multiple nodes can have read access or one node write access (multiple readers-one writer protocol)
    - After a write, all copies are invalidated or updated
    - DSM has to keep track of locations of all copies of data objects. Examples of implementations:
        - IVY: owner node of data object knows all nodes that have copies
        - PLUS: distributed linked-list tracks all nodes that have copies

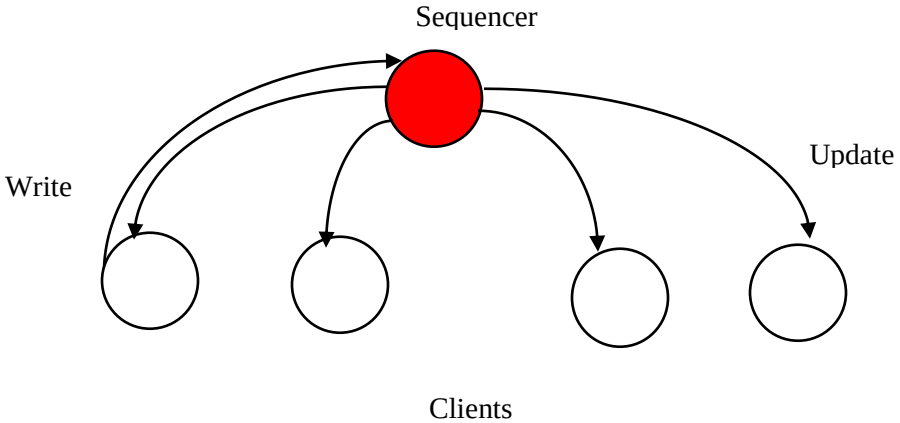
– Advantage

- The read-replication can lead to substantial performance improvements if the ratio of reads to writes is large



4. **The Full–Replication Algorithm**

- Extension of read-replication algorithm: multiple nodes can read and multiple nodes can write (multiple-readers, multiple-writers protocol)
- Issue: consistency of data for multiple writers
- Solution: use of gap-free sequencer
  - All writes sent to sequencer
  - Sequencer assigns sequence number and sends write request to all sites that have copies
  - Each node performs writes according to sequence numbers
  - A gap in sequence numbers indicates a missing write request: node asks for retransmission of missing write requests



Online Courseware for B.Tech. Computer Science and Engineering Program(Autonomy)
Paper Name: Distributed Operating System
Paper Code: IT(CS)605B

Module 4: Case Study

Lecture 1: AMOEBA: Introduction, Process management

Twenty years ago computers were large and expensive, so many users had to share a single computer. This was the age of the mainframes and minicomputers. Ten years ago it became economically feasible to give each user his own computer, which led to the concept of personal computing. Prices have continued to drop, so it is now feasible to have many CPUs per user. The question is: how should such a system be organized? In particular, what should the software be like.

One possible structure is to give each user a personal multiprocessor, with 10, 20, or more CPUs. However, in such a system, most of the CPUs will be idle most of the time, leading to inefficient use of resources, especially when some users may need large bursts of computing power at random intervals. Instead, we envision a different model, one consisting of a large rack of single-board computers (the processor pool) located in the machine room, plus user terminals for accessing the system. All these machines are to be connected by a high-speed network.Pool processors are inherently cheaper than workstations because they consist of just a single board with a network connection. There is no keyboard, monitor, or mouse, and the power supply can be shared by many boards. Since the pool processors are allocated only when needed, an idle user only ties up an inexpensive X terminal instead of an expensive workstation.

Amoeba was designed as the operating system for such a system. The Amoeba user can loginto the system as a whole, and use it without having to think about which process is running where.To the user, Amoeba is a single integrated system, not as a loose collection of machines connected bya network. In particular, Amoeba has no concept of a ‘‘home machine’’ on which the user normallyworks, with occasional requests made to other machines. Instead, when a user logs in, a shell isstarted somewhere, on a machine started by the system. Subsequent commands are run on one ormore machines chosen by the system. The user’s workstation simply functions as an X-terminal toallow access to the Amoeba resources. This model is fundamentally different from client-servercomputing used at many installations, in which the user’s machine is a client that makes requests ofvarious servers from time to time.In terms of software, an Amoeba system consists of processes and objects. Processes performwork, and communicate with each other by a structured form of message passing. Services and information in Amoeba are organized and protected as objects. Typical objects include processes,memory segments, files, and directories. Each object has a capability that controls it. A processholding the capability for an object can invoke its methods. Capabilities are protected cryptographically, to prevent processes from forging capabilities.

The amoeba system architecture

The hardware of an Amoeba system has three principal components all connected by a LAN:
- An X-terminal or workstation running X-Windows for each user
- The processor pool (a rack of single-board computers)
- A certain number of dedicated servers (file server, etc.)

Each user has an X-terminal (or workstation running X-Windows) for talking to the system.Although it is technically possible to run user jobs on this machine, that is not normally done, as it isthe intention to have Amoeba programs use multiple processors in parallel to improve performance.

The processor pool idea is based on the assumption that the ratio of processors to users is large.Currently at the VU we typically have 10 users and a pool with 80 processors. As time goes on, theratio of machines to people will increase. The current personal computer model does not address thistrend well, which is why we have devised an alternative model for the future.

In addition to the rack of identical pool processors, a small number of dedicated servers providecertain important services. Aserviceis an abstract definition of what the server is prepared to do forits clients. This definition defines what the client can ask for and what the results will be, but it doesnot specify how many servers are working together to provide the service. In this way, the systemhas a mechanism for providing fault-tolerant services by having multiple servers doing the work.

An example is the directory server. There is nothing inherent about the directory server or thesystem design that would prevent a user from starting up a new directory server on a pool processorevery time he wanted to look up a file name. However, doing so would be horrendously inefficient,so one or more directory servers are kept running all the time, generally on dedicated machines toenhance their performance. The decision to have some servers always running and others to bestarted explicitly when needed is up to the system administrator.

Lecture 2: AMOEBA-Communication

Communication in amoeba

Amoeba supports two forms of communication: remote procedure call (RPC) using point-to-point message passing, and group communication. At the lowest level, an RPC consists of a request message followed by a reply message. Group communication uses hardware broadcasting or multicasting if it is available; otherwise, the kernel transparently simulates it with individual messages. Inthis section we will describe both Amoeba RPC and Amoeba group communication.

Normal point-to-point communication in Amoeba consists of a client sending a message to aserver followed by the server sending a reply back to the client. The RPC primitive that sends therequest automatically blocks the caller until the reply comes back, thus forcing a certain amount ofstructure on programs. Separatesendandreceiveprimitives can be thought of as the distributedsystem's answer to thegotostatement: parallel spaghetti programming. They should be avoided byuser programs and used only by language runtime systems that have unusual communication requirements.

Each standard server defines a procedural interface that clients can call. These library routinesare stubs that pack the parameters into messages and invoke the kernel primitives to send the message. During message transmission, the stub, and hence the calling thread, are blocked. When thereply comes back, the stub returns the status and results to the client.

In order for a client thread to do an RPC with a server thread, the client must know the server'saddress. Addressing is done by allowing any thread to choose a random 48-bit number, called aport,to be used as the address for messages sent to it. Different threads in a process may use differentports if they so desire. All messages are addressed from a sender to a destination port. A port isnothing more than a kind of logical thread address. There is no data structure and no storage associated with a port. It is similar to an IP address or an Ethernet address in that respect, except that it isnot tied to any particular physical location.

RPC is not the only form of communication supported by Amoeba. It also supports group communication. A group in Amoeba consists of one or more processes that are cooperating to carry outsome task or provide some service. Amoeba works best on LANs that support either multicasting orbroadcasting (or like Ethernet, both). For simplicity, we will just refer to broadcasting, although infact the implementation uses multicasting when it can to avoid disturbing machines that are notinterested in the message being sent. It is assumed that the

hardware broadcast is good, but not perfect. In practice, lost packets are rare, but receiver overruns do happen occasionally. Since theseerrors can occur they cannot simply be ignored, so the protocol has been designed to deal with them.

The key idea that forms the basis of the implementation of group communication isreliablebroadcasting. By this we mean that when a user process broadcasts a message the user-suppliedmessage is delivered correctly to all members of the group, even though the hardware may lose packets. Central to the algorithm is a process called thesequencer, which numbers broadcasts in order.When an application process executes a broadcasting primitive, a trap to its kernel occurs. The kernel sends the message to the sequencer using a normal point-to-point message.

Now consider what happens at the sequencer when aRequest for Broadcastpacket arrivesthere. If the message is new (normal case), the next sequence number is assigned to it, and thesequencer counter incremented by 1 for next time. The message and its identifier are then stored in ahistory buffer, and the message is then broadcast. If the packet is a retransmission, it is ignored.Finally, let us consider what happens when a kernel receives a broadcast. First, the sequencenumber is compared to the sequence number of the broadcast received most recently. If the new oneis 1 higher (normal case), no broadcasts have been missed, so the message is passed up to the application program, assuming that it is waiting. If it is not waiting, it is buffered until the program calls*ReceiveFromGroup*.

Suppose that the newly received broadcast has sequence number 25, while the previous one hadnumber 23. The kernel is immediately alerted to the fact that it has missed number 24, so it sends apoint-to-point message to the sequencer asking for a private retransmission of the missing message.The sequencer fetches the missing message from its history buffer and sends it. When it arrives, thereceiving kernel processes 24 and 25, passing them to the application program in numerical order.Thus the only effect of a lost message is a (normally) minor time delay. All application programs seeall broadcasts in the same order, even if some messages are lost.Several variations of this algorithm also exist. In one, the process wanting to send a broadcastjust does so, and the sequencer just broadcasts an OK message giving its sequence number. Inanother variant, processes first ask the sequencer for a number, then use this number in their ownbroadcasts. These variants make different tradeoffs between bandwidth and interrupts.

Process management in amoeba

A process in Amoeba is basically an address space and a collection of threads that run in it. Aprocess with one thread is roughly analogous to a UNIX process in terms of how it behaves and whatit can do. A process is an object in Amoeba. When a process is created, the parent

process is given acapability for the child process, just as with any other newly created object. Using this capability, thechild can be suspended, restarted, signaled, or destroyed.
Process management is handled at three different levels in Amoeba. At the lowest level are the process servers, which are kernel threads running on every machine. To create a process on a given
machine, another process does an RPC with that machine's process server, providing it with thenecessary information. At the next level up we have a set of library procedures that provide a moreconvenient interface for user programs. Several flavors are provided. They do their job by callingthe low-level interface procedures. Finally, the simplest way to create a process is to use the runserver, which does most of the work of determining where to run the new process.
Some of the process management calls use a data structure called aprocess descriptor to provide information about the process to be run. One field in the process descriptor tells which CPUarchitecture the process can run on. In heterogeneous systems, this field is essential to make sure thatPentium binaries are not run on SPARCs, and so on. Another field contains the process' owner'scapability. When the process terminates or is stunned (see below), RPCs will be done using thiscapability to report the event. It also contains descriptors for all the process' segments, which collectively define its address space, as well as descriptors for all its threads. Finally, the process descriptoralso contains a descriptor for each thread in the process. The content of a thread descriptor is architecture dependent, but as a bare minimum, it contains the thread's program counter and stack pointer.

Amoeba supports a simple threads model. When a process starts up, it has one thread. Duringexecution, the process can create additional threads, and existing threads can terminate. The numberof threads is therefore completely dynamic. When a new thread is created, the parameters to the callspecify the procedure to run and the size of the initial stack. Although all threads in a process sharethe same program text and global data, each thread has its own stack, its own stack pointer, and itsown copy of the machine registers.

Three methods are provided for threads to synchronize: signals, mutexes, and semaphores. Signals are asynchronous interrupts sent from one thread to another thread in the same process. A mutexis like a binary semaphore. General semaphores are also provided.

All threads are managed by the kernel. The advantage of this design is that when a thread doesan RPC, the kernel can block that thread and schedule another one in the same process if one is ready.Thread scheduling is done using priorities, with kernel threads getting higher priority than userthreads. Thread scheduling can be set up to be either pre-emptive or run-to-completion (i.e., threadscontinue to run until they block), as the process wishes.

Lecture 3: MACH - Introduction

Introduction

The Mach project [Acetta et al. 1986, Loepere 1991, Boykin et al. 1993] was based at Carnegie Mellon University in the USA until 1994. Its development into a real-time kernel continued there[Lee et al. 1996], and groups at the University of Utah and the Open Software Foundationcontinued its development. The Mach project was successor to two other projects, RIG [Rashid1986] and Accent [Rashid and Robertson 1981, Rashid 1985, Fitzgerald and Rashid 1986]. RIGwas developed at the University of Rochester in the 1970s, and Accent was developed at Carnegie Mellon during the first half of the 1980s. In contrast to its RIG and Accent predecessors, the Machproject never set out to develop a complete distributed operating system. Instead, the Mach kernelwas developed to provide direct compatibility with BSD UNIX. It was designed to provideadvanced kernel facilities that would complement those of UNIX and allow a UNIXimplementation to be spread across a network of multiprocessor and single-processor computers.From the beginning, the designers' intention was for much of UNIX to be implemented as user-level processes.

Despite these intentions, Mach version 2.5, the first of the two major releases, included allthe UNIX compatibility code inside the kernel itself. It ran on SUN-3s, the IBM RT PC,multiprocessor and uniprocessor VAX systems, and the Encore Multimax and Sequentmultiprocessors, among other computers. From 1989, Mach 2.5 was incorporated as the basetechnology for OSF/1, the Open Software Foundation's rival to System V Release 4 as theindustry-standard version of UNIX. An older version of Mach was used as a basis for the operatingsystem for the NeXT workstation.The UNIX code was removed from the version 3.0 Mach kernel, however, and it is thisversion that we describe. Most recently, Mach 3.0 is the basis of the implementation of MkLinux,a variant of the Linux operating system running on Power Macintosh computers [Morin 1997]. Theversion 3.0 Mach kernel also runs on Intel x86-based PCs. It ran on the DECstation 3100 and 5000series computers, some Motorola 88000-based computers and SUN SPARCStations; ports wereundertaken for IBM's RS6000, Hewlett-Packard's Precision Architecture and Digital EquipmentCorporation's Alpha.

Version 3.0 Mach is a basis for building user-level emulations of operating systems, databasesystems, language run-time systems and other items of system software that we call subsystems.The emulation of conventional operating systems makes it possible to run existing binariesdeveloped for them. In addition, new applications for these conventional operating

systems can bedeveloped. At the same time, middleware and applications that take advantage of the benefits ofdistribution can be developed; and the implementations of the conventional operating systems canalso be distributed. Two important issues arise for operating system emulations. First, distributedemulations cannot be entirely accurate, because of the new failure modes that arise withdistribution. Second, the question is still open of whether acceptable performance levels can beachieved for widespread use.

System Components

To achieve the design goals of Mach, the developers reduced the operating system functionality to small set of basic abstractions, out of which all otherfunctionality can be derived. The Mach approach is to place as little as possiblewithin the kernel but to make what is there powerful enough that all otherfeatures can be implemented at the user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel,and all data movement among processes, are handled through one communication mechanism. Mach is therefore able to provide system-wide protectionto its users by protecting the communications mechanism. Optimizing thisone communications path can result in significant performance gains, and it issimpler than trying to optimize several paths. Mach is extensible, because manytraditionally kernel-based functions can be implemented as user-level servers.For instance, all pagers (including the default pager) can be implementedexternally and called by the kernel for the user.

Mach is an example of an object-oriented system where the data andthe operations that manipulate that data are encapsulated into an abstractobject. Only the operations of the object are able to act on the entities definedin it. The details of how these operations are implemented are hidden, asare the internal data structures. Thus, a programmer can use an object onlyby invoking its defined, exported operations. A programmer can change theinternal operations without changing the interface definition, so changes andoptimizations do not affect other aspects of system operation. The object oriented approach supported by Mach allows objects to reside anywhere ina network of Mach systems, transparent to the user. Theportmechanism,discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system and are as follows:

- Ataskis an execution environment that provides the basic unit of resourceallocation. It consists of a virtual address space and protected access tosystem resources via ports, and it may contain one or more threads.
- Athreadis the basic unit of execution and must run in the context of atask (which provides the address space). All threads within a task sharethe tasks' resources (ports, memory, and so on). There is no notion of aprocessin Mach. Rather, a traditional process would be implemented as ataskwithasinglethreadofcontrol.

- Aportis the basic object-reference mechanism in Mach and is implemented as a kernel-protected communication channel. Communication isaccomplished by sending messages to ports; messages are queued at thedestination port if no thread is immediately ready to receive them. Portsare protected by kernel-managed capabilities, orport rights;ataskmusthave a port right to send a message to a port. The programmer invokes anoperationonanobjectbysendinga message to a port associated with thatobject. The object being represented by a portreceivesthe messages.
- Aport setis a group of ports sharing a common message queue. A threadcan receive messages for a port set and thus service multiple ports. Eachreceived message identifies the individual port (within the set) from whichit was received; the receiver can use this to identify the object referred toby the message.
- Amessageis the basic method of communication between threads inMach. It is a typed collection of data objects; for each object, it may containthe actual data or a pointer to out-of-line data. Port rights are passed inmessages; this is the only way to move them among tasks. (Passing a portright in shared memory does not work, because the Mach kernel will notpermit the new task to use a right obtained in this manner.)
- Amemory objectis a source of memory; tasks can access it by mappingportions of an object (or the entire object) into their address spaces. Theobject can be managed by a user-mode external memory manager. Oneexample is a file managed by a file server; however, a memory object canbe any object for which memory-mapped access makes sense. A mappedbuffer implementation of aUNIXpipe is one example.

Process Management

A task can be thought of as a traditional process that does not have aninstruction pointer or a register set. A task contains a virtual-address space, aset of port rights, and accounting information. A task is a passive entity thatdoes nothing unless it has one or more threads executing in it.

Basic Structure

A task containing one thread is similar to aUNIXprocess. Just as aforksystemcall produces a newUNIXprocess, Mach creates a new task to emulate thisbehavior. The new task's memory is a duplicate of the parent's address space,as dictated by theinheritance attributesof the parent's memory. The new taskcontains one thread, which is started at the same point as the creatingforkcallin the parent. Threads and tasks can also be suspended and resumed.Threads are especially useful in server applications, which are commoninUNIX, since one task can have multiple threads to service multiple requeststo the task. Threads also allow efficient use of parallel computing resources.Rather than having one process on each processor, with the correspondingperformance penalty and operating-system overhead, a task can have itsthreads spread among parallel processors. Threads add efficiency to user-levelprograms as well. For instance, inUNIX, an entire process must wait when apage fault occurs or when a system call is

executed. In a task with multiplethreads, only the thread that causes the page fault or executes a system callis delayed; all other threads continue executing. Because Mach has kernel-supported threads (Chapter 4), the threads have some cost associated withthem. They must have supporting data structures in the kernel, and morecomplex kernel-scheduling algorithms must be provided. These algorithmsand thread states are discussed in Chapter 4.

At the user level, threads may be in one of two states:

- Running: The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).

- Suspended: The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state. These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can bebuilt. Thispractice isconsistent withMach’sphilosophyofprovidingminimumyet sufficient functionality in the kernel. The MachIPCfacility can be used forsynchronization, with processes exchanging messages at rendezvous points.Thread-level synchronization is provided by calls to start and stop threads atappropriate times. Asuspend countis kept for each thread. This count allowsmultiplesuspendcalls to be executed on a thread, and only when an equalnumber ofresumecalls occur is the thread resumed. Unfortunately, this featurehas its own limitation. Because it is an error for astartcall to be executedbefore astopcall (thesuspend countwould become negative), these routinescannot be used to synchronize shared data access. However,waitandsignaloperations associated with semaphores, and used for synchronization, can beimplemented via theIPCcalls.

Lecture 4: MACH - Communication

Communication model

Mach provides a single system call for message passing: mach_msg. Before describing this, weshall say more about messages and ports in Mach.

Messages: A message consists of a fixed-size header followed by a variable-length list of data items The fixed-size header contains:

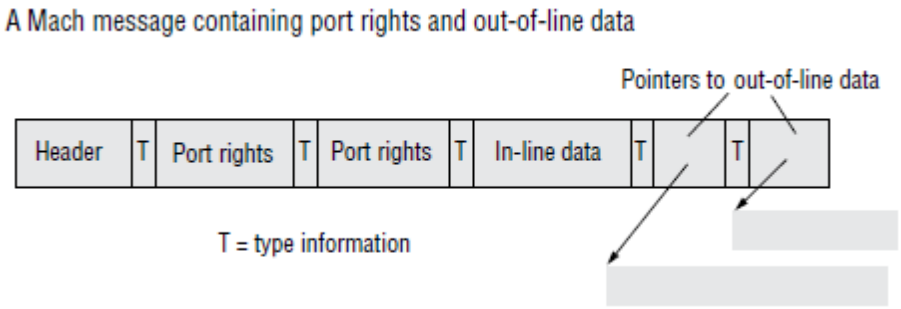
The destination port: For simplicity, this is part of the message rather than being specified asa separate parameter to the mach_msgsystem call. It is specified by the local identifier of theappropriate send rights.

A reply port: If a reply is required, then send rights to a local port (that is, one for which thesending thread has receive rights) are enclosed in the message for this purpose.

An operation identifier: This identifies an operation (procedure) in the service interface andis meaningful only to applications.

Extra data size: Following the header (that is, contiguous with it) there is, in general, a variable sized list of typed items. There is no length limit to this, except the number of bitsin this field and the total address space size.Each item in the list after the message header is one of the following (which can occur in any orderin the message):

Typed message data: individual, in-line type-tagged data items;
Port rights: referred to by their local identifiers;
Pointers to out-of-line data: data held in a separate non-contiguous block of memory.



Ports

A Mach port has a message queue whose size can be set dynamically by the task with receiverights. This facility enables receivers to implement a form of flow control. When a normal send
right is used, a thread attempting to send a message to a port whose message queue is full will be blocked until room becomes available. When a thread uses a send-once right, the recipient always
queues the message, even if the message queue is full. Since a send-once right is used, it is known
that no further messages can be sent from that source. Server threads can avoid blocking by using
send-once rights when replying to clients.

Sending port rights - When port send rights are enclosed in a message, the receiver acquires send
rights to the same port. When receive rights are transmitted, they are automatically de-allocated in
the sending task. This is because receive rights cannot be possessed by more than one task at a time.
All messages queued at the port and all subsequently transmitted messages can be received by the
new owner of receive rights, in a manner that is transparent to tasks sending to the port. Thetransparent transfer of receive rights is relatively straightforward to achieve when the rights aretransferred within a single computer. The acquired capability is simply a pointer to the localmessage queue.

Monitoring connectivity - The kernel is designed to inform senders and receivers when conditionsarise under which sending or receiving messages would be futile. For this purpose, it keepsinformation about the number of send and receive rights referring to a given port. If no task holdsreceive rights for a particular port (for example, because the task holding these rights failed), thenall send rights in local tasks' port name spaces become *dead names*. When a sender attempts to se
a name referring to a port for which receive rights no longer exist, the kernel turns the name into adead name and returns an error indication. Similarly, tasks can request the kernel to notify themasynchronously of the condition that no send rights exist for a specified port. The kernel performsthis notification by sending the requesting thread a message, using send rights given to it by thethread for this purpose. The condition of no send rights can be ascertained by keeping a referencecount that is incremented whenever a send right is created and decremented when one is destroyed.It should be stressed that the conditions of no senders/no receiver are tackled within thedomain of a single kernel at relatively little cost. Checking for these conditions in a distributedsystem is, by contrast, a complex and expensive operation. Given that rights can be sent inmessages, the send or receive rights for a given port could be held by any task, or even be in amessage, queued at a port or in transit between computers.

Port sets - Port sets are locally managed collections of ports that are created within a single task.

When a thread issues a receive from a port set, the kernel returns a message that was delivered tosome member of the set. It also returns the identifier of this port's receive rights so that the threadcan process the message accordingly.Ports sets are useful because typically a server is required to service client messages at all ofits ports at all times. Receiving a message from a port whose message queue is empty blocks athread, even if a message that it could process arrives on another port first. Assigning a thread toeach port overcomes this problem but is not feasible for servers with large numbers of portsbecause a thread is a more expensive resource than a port. By collecting ports into a port set, asingle thread can be used to service incoming messages without fear of missing any. Furthermore,this thread will block if no messages are available on any port in the set, so avoiding a busy-waitingsolution in which the thread polls until a message arrives on one of the ports.

Mach_msg

The *Mach_msg* system call provides for both asynchronous message passing and request-reply-styleinteractions, which makes it extremely complicated. The complete call is as follows:

mach_msg(msg_header, option, snd_siz, rcv_siz, rcv_name, timeout, notify)msg_header points to a common message header for the sent and received messages, *option*specifies send, receive or both, *snd_siz* and *rcv_siz* give the sizes of the sent and receivedmessage buffers, *rcv_name* specifies the port or port set receive rights (if a message is received),*timeout* sets a limit to the total time to send and/or receive a message, *notify* supplies port rightswhich the kernel is to use to send notification messages under exceptional conditions.

Lecture 5: DCE - Introduction, Process management

DCE (Distributed Computing Environment) is an architecture defined by the Open Software Foundation (OSF) to provide an Open Systems platform to address the challenges of distributed computing. It is being ported to all major IBM(R) and many non-IBM environments. Note that all current DCE implementations use TCP/IP rather than SNA as their communication protocol.

The Distributed Computing Environment (DCE) is a software system developed in the early 1990s by a consortium that included Apollo Computer (later part of Hewlett-Packard), IBM, Digital Equipment Corporation, and others. The DCE supplies a framework and toolkit for developing client/server applications. The framework includes a remote procedure call (RPC) mechanism known as DCE/RPC, a naming (directory) service, a time service, an authentication service, an authorization service and a distributed file system (DFS) known as DCE/DFS.

Architecture

The largest unit of management in DCE is a **cell**. The highest privileges within a cell are assigned to a role called *cell administrator*, normally assigned to the "user" *cell_admin*. Note that this need not be a real OS-level user. The cell_admin has all privileges over all DCE resources within the cell. Privileges can be awarded to or removed from the following categories : user_obj, group_obj, other_obj, any_other for any given DCE resource. The first three correspond to the owner, group member, and any other DCE principal respectively. The last group contains any non-DCE principal. Multiple cells can be configured to communicate and share resources with each other. All principals from external cells are treated as "foreign" users and privileges can be awarded or removed accordingly. In addition to this, specific users or groups can be assigned privileges on any DCE resource, something which is not possible with the traditional UNIX filesystem, which lacks ACLs.

Major components of DCE within every cell are:

1. the **security server** that is responsible for authentication
2. The **Cell Directory Server** (CDS) that is the respository of resources and ACLs and
3. The **Distributed Time Server** that provides an accurate clock for proper functioning of the entire cell. Modern DCE implementations such as IBM's are fully capable of interoperating with Kerberos as the security server, LDAP for the CDS and the Network Time Protocol implementations for the time server.

Lecture 6: DCE-Communication.

DCE is based on three distributed computing models:

Client/server

A way of organizing a distributed application

Remote procedure call

A way of communicating between parts of a distributed application

Shared files

A way of handling data in a distributed system, based on a personal computer file access model.

Remote procedure call (RPC)

One way of implementing communications between a client and a server of a distributed application is to use the procedure call model. In this model, the client makes what looks like a procedure call, and waits for a reply from the server. The procedure call is translated into network communications by the underlying RPC mechanism. The server receives a request and executes the procedure, returning the results to the client.

In DCE RPC, you define one or more DCE RPC interfaces, using the DCE interface definition language (IDL). Each interface comprises a set of associated RPC calls (called *operations*), each with their input and output parameters. You compile the IDL, which generates data structure definitions and executable stubs for both the client and the server. The matching parameter data structures ensure a common view of the parameters by both client and server. The matching client and server executable stubs handle the necessary data transformations to and from the network transmission format, and between different machine formats (EBCDIC and ASCII).

You use the DCE Directory Service to advertise that your server now supports the new interface you defined using the IDL. Your client code can likewise use the Directory Service to discover which servers provide the required interface.

You can also use the DCE Security Service to ensure that only authorized client end users can access your newly defined server function.

Reference

1. https://cds.cern.ch/record/400320/files/p109.pdf
2. Pradeep K. Sinha-Distributed Operating Systems_ Concepts and Design-P K Sinha
3. Distributed operating systems - Tanenbaum Andrew
4. https://middlewares.wordpress.com/2007/12/28/introduction-to-the-distributed-computing-environment/