SATHYABAMA UNIVERSITY

(Established under Section 3, UGC Act 1956)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

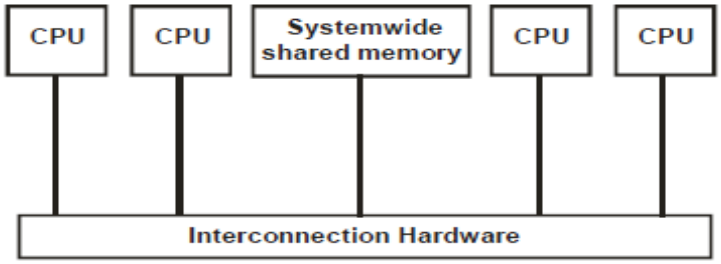
SCSX 1028 DISTRIBUTED COMPUTING

SCSX 1028 DISTRIBUTED COMPUTING

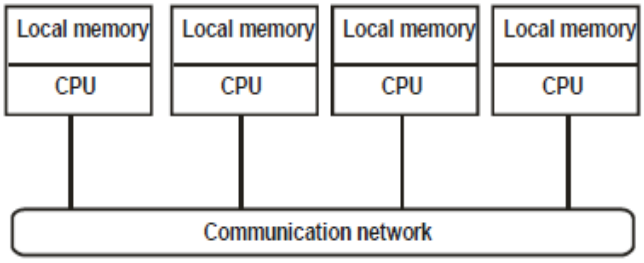
UNIT 1 – FUNDAMENTALS

WHAT IS A DISTRIBUTED COMPUTING SYSTEM?

- Advancements in microelectronic technology have resulted in the availability of fast, inexpensive processors and advancements in communication technology.
- These advancements have resulted in the availability of cost effective and highly efficient computer networks.
- The net result of these advancements replaced single, high speed processor into interconnected, multiple processors.
- 2 types of interconnected, multiple processors are
    - Tightly coupled systems – single system wide primary memory (address space) that is shared by all the processors. Communication between the processors usually takes place through the shared memory.



    - Loosely coupled systems – the processors do not share memory and each processor has its own local memory. All physical communication between the processors are done by passing messages across the network that interconnects the processors.



- Usually tightly coupled systems are referred to as parallel processing systems and loosely coupled systems are referred to as distributed computing systems or simply distributed systems.

- Distributed systems are more freely expandable and can have an almost unlimited number of processors.
- Definition: A distributed computing system is basically a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals and the communication between any two processors of the system takes place by message passing over the communication network.
- For a particular processor, its own resources are local whereas the other processors and their resources are remote.
- A processor and its resources are usually referred to as a node or site or machine of the distributed computing system.

EVOLUTION OF DISTRIBUTED COMPUTING SYSTEMS

- From 1945, when the modem Computer era began, until about 1985, computers were large and expensive.
- Even minicomputers cost at least tens of thousands of dollars each. As a result, most organizations had only a handful of computers, and for lack of a way to connect them, these operated independently from one another.
- Starting around the mid – 1980s, however, two advances in technology began to change that situation.
- The first was the development of **powerful microprocessors**. Initially, these were 8-bit machines, but soon 16, 32, and 64-bit CPUs became common. Many of these had the computing power of a mainframe (i.e., large) computer, but for a fraction of the price.
- The second development was the invention of **high-speed computer networks**. Local-area networks or LANs allow hundreds of machines within a building to be connected in such a way that small amounts of information can be transferred between machines in a few microseconds or so.
- Larger amounts of data can be **Distributed Computing** become popular with the difficulties of centralized processing in mainframe use.
- With mainframe software architectures, all components are within a central host computer. Users interact with the host through a terminal that captures keystrokes and sends that information to the host.

- In the last decade, however mainframes have found a new use as a server in distributed **client/server architectures**. The original PC networks were based on file sharing architectures, where the server transfers files from a shared location to a desktop environment.
- **File sharing architectures** work well if shared usage is low, update contention is low, and the volume of data to be transferred is low.
- In the 1990s, PC LAN (Local Area Network) computing changed because the capacity of the file sharing was strained as the number of online users grew and graphical user interfaces (GUIs) became popular.
- The next major step in distributed computing came with separation of software architecture into **2 or 3 tiers**.
- With **two tier client-server architectures**, the GUI is usually located in the user's desktop environment and the database management services are usually on a server that is a more powerful machine that services many clients.
- Processing management is split between the user system interface environment and the database management server environment. The two tier client/server architecture is a good solution for locally distributed computing (Dozen to 100 people interacting on a LAN simultaneously).
- When the number of users exceeds 100, performance begins to deteriorate and the architecture is also difficult to scale. The **three tier architecture** (also referred to as the multi-tier architecture) emerged to overcome the limitations of the two tier architecture.
- In the three tier architecture, a middle tier was added between the user system interface client environment and the database management server environment.
- There are a variety of ways of implementing this middle tier, such as transaction processing monitors, messaging middleware, or application servers. The middle tier can perform queuing, application execution, and database queries.
- Three tier architecture proved successful at separating the logical design of systems, the complexity of collaborating interfaces was still relatively difficult due to technical dependencies between interconnecting processes. Standards for **Remote Procedure Calls** (RPC) were then used as an attempt to standardize interaction between processes.
- As an interface for software to use it is a set of rules for marshalling and un-marshalling parameters and results, a set of rules for encoding and decoding information transmitted between two processes; a few primitive operations to invoke an individual call, to return its

results, and to cancel it. RPC requires a communications infrastructure to set up the path between the processes and provide a framework for naming and addressing.

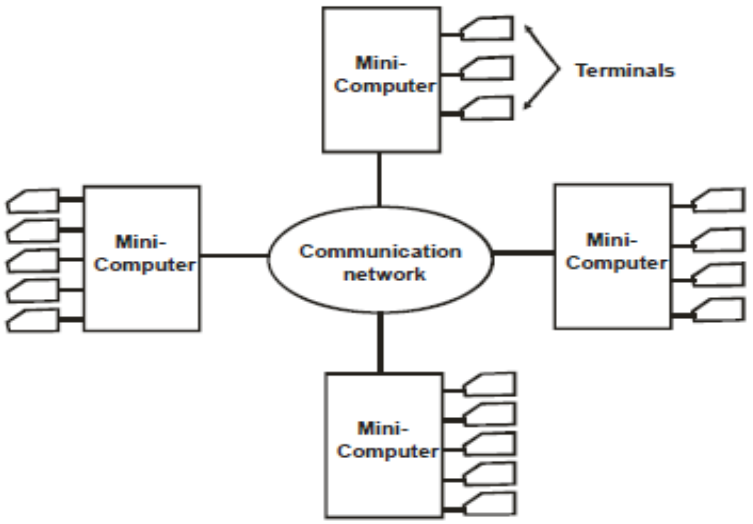
- There are two models that provide the framework for using the tools. These are known as the **computational model and the interaction model**. The computational model describes how a program executes a procedure call when the procedure resides in a different process. The interaction model describes the activities that take place as the call progresses.
- A marshalling component and a encoding component are brought together by an **Interface Definition Language** (IDL). An IDL program defines the signatures of RPC operations.

DISTRIBUTED COMPUTING SYSTEM MODELS

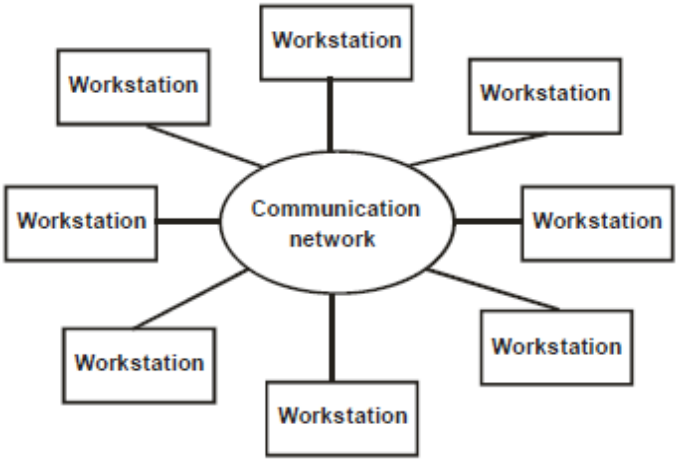
- Various Models are used to build distributed computing systems. These models can be broadly classified into five categories.
    - Minicomputer Model
    - Workstation Model
    - Workstation Server Model
    - Processor Pool Model
    - Hybrid Model

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 onto 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.
- A company's office or a university department may have several workstations scattered 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 (especially at night), 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 on to other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently.

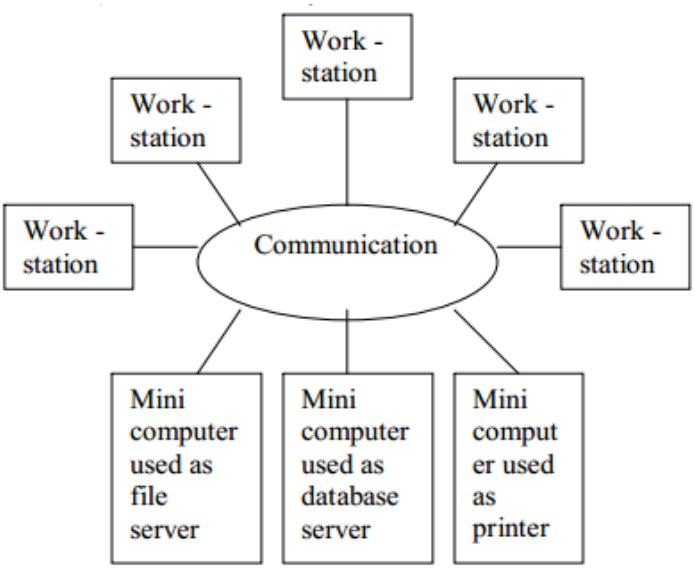
- In this model, a user logs onto one of the workstations called his or her "home" workstation and submits jobs for execution. When the system finds that the user's workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the process from the user's workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user's workstation.
- This model is not so simple to implement as it might appear at first sight because several issues must be resolved. These issues are
  1. How does the system find an idle workstation?
  2. How is a process transferred from one workstation to get it executed on another workstation?
  3. What happens to a remote process if a user logs on to a workstation that was idle until now and was being used to execute a process of another workstation?
- Three commonly used approaches for handling the third issue are as follows:
  1. The first approach is to allow the remote process share the resources of the workstation along with its own logged-on user's processes. This method is easy to implement, but it defeats the main idea of workstations serving as personal computers.
  2. The second approach is to kill the remote process. The main drawbacks of this method are that all processing done in the remote process gets lost and the file system may be left in an inconsistent state, making this method unattractive.
  3. The third approach is to migrate the remote process back to its home workstation, so that its execution can be continued there. This method is difficult to implement because it requires the system to support preemptive process migration facility.
- 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.

Workstation Server Model:

- 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, we need new model called workstation server model.
- In this model, a user logs on to a workstation called his or her home workstation. Normal computation activities required by the user's processes are preformed 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 be migrated to the server machines for getting the work done by those machines.
- For better overall system performance, the local disk of a diskful workstation is normally used for such purposes as storage of temporary files, storage of unshared files, storage of shared files that are rarely changed, paging activity in the virtual - memory management, and changing of remotely accessed data.



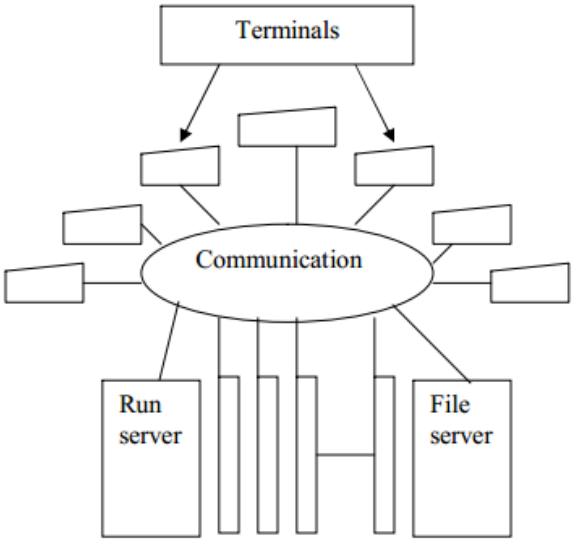
- As compared to the workstation model, the workstation server model has several advantages:
  1. In general, it is much cheaper to use a few minicomputers equipped with large, fast disks that are accessed over the network than a large number of diskful workstations, with each workstation having a small, slow disk.
  2. Diskless workstations are also preferred to diskful workstations from a system maintenance point of view. Backup and hardware maintenance are easier to perform with a few larger disks than with many small disks scattered all over a building or campus. Furthermore, installing new releases of software is easier when the software is to be installed on a few file server machines than on every workstation.
  3. In the workstation server model, since all files are managed by the file servers, users have the flexibility to use any workstation and access the files in the same manner irrespective of which workstation the user is currently logged on.

4. In the workstation server model, the request response protocol described above is mainly used to access the services of the server machines. Therefore, unlike the workstation model, this model does not need a process migration facility, which is difficult to implement.
    - The request response protocol is known as the client-server model of communication. In this model, a client process sends a request to a server process for getting some service such as a block of a file. The server executes the request and sends back a reply to the client that contains the result of request processing.
    - The client-server model provides an effective general – purpose approach to the sharing of information and resources in distributed computing systems.
5. A user has guaranteed response time because workstations are not used for executing remote processes. However, the model does not utilize the processing capability of idle workstations.

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 he or she 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.
- As shown in the figure, in the pure processor-pool model, the processors in the pool have no terminals attached directly to them, and users access the system from terminals that are attached to the network via special devices.
- These terminals are either small diskless workstations or graphic terminals, such as X terminals. A special server (Called a Run server) manages and allocates the processors in the pool to different users on a demand basis. When a user submits a job for computation, an appropriate number of processors are temporarily assigned to his or her job by the run server.

- In the processor-pool model, there is no concept of a home machine. That is, a user does not log onto a particular machine, but the system as a whole.



Hybrid Model:

- Out of the four models described above, the workstation server model, is the most widely used models 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.
- To continue the advantages of both the workstation server and processor pool models, a hybrid model may be used to build a distributed computing system.
- The hybrid model is based on the workstation server model, but with the addition of a pool of processors. The processors in the pool can be allocated dynamically for computations that are too large for workstations or that requires several computers concurrently for efficient execution.
- In addition to efficient execution of computation-intensive jobs, the hybrid model gives guaranteed response to interactive jobs by allowing them to be processed on local workstations of the users. However, the hybrid model is more expensive to implement than the workstation – server model or the processor-pool model.

WHY ARE DISTRIBUTED COMPUTING SYSTEMS GAINING POPULARITY?

The following are the reasons for the popularity of Distributed Computing Systems

1. Inherently Distributed Applications
2. Information Sharing among Distributed Users
3. Resource Sharing
4. Better Price Performance Ratio
5. Shorter Response Times and Higher Throughput
6. Higher Reliability
7. Extensibility and Incremental Growth
8. Better Flexibility in Meeting Users' Needs

WHAT IS A DISTRIBUTED OPERATING SYSTEM?

- An operating system is a program that controls the resources of a computer system and provides its users with an interface or virtual machine that is more convenient to use than the bare machine. According to this definition, the two primary tasks of an operating system are as follows
    1. To present users with a virtual machine that is easier to program than the underlying hardware.
    2. To manage the various resources of the system. This involves performing such tasks as keeping track of who is using which resource, granting resource requests, accounting for resource usage, and mediating conflicting requests from different programs and users.
- The operating systems commonly used for distributed computing systems can be broadly classified into two types-network operating systems and distributed operating systems. The three most important features commonly used to differentiate between these two types of operating systems are
    - System image
    - Autonomy
    - Fault Tolerance capability.

System image:

- The most important feature used to differentiate between the two types of operating systems is the image of the distributed computing system from the point of view of its users.

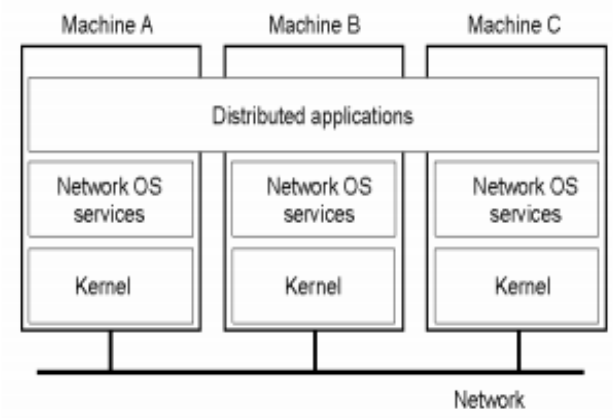
- In case of a network operating system, the users view the distributed computing system as a collection of distinct machines connected by a communication subsystem.
- A distributed operating system hides the existence of multiple computers and provides a single-system image to its users. That is, it makes a collection of networked machines act as a virtual uniprocessor.

Autonomy:

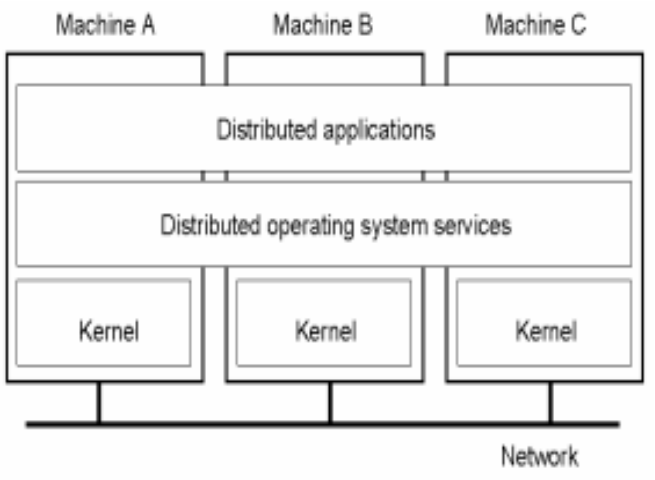
- In the case of a network operating system, each computer of the distributed computing system has its own local operating system and there is essentially no coordination at all among the computers except for the rule that when two processes on different computers communicate with each other, they must use a mutually agreed on communication protocol.
- With a distributed operating system, there is a single system wide operating system and each computer of the distributed computing system runs a part of this global operating system. The distributed operating system tightly interweaves all the computers of the distributed computing system in the sense that they work in close cooperation with each other for the efficient and effective utilization of the various resources of the system.

Fault tolerance capability:

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



Network Operating System



Distributed Operating System

ISSUES IN DESIGNING A DISTRIBUTED OPERATING SYSTEM

- In general, designing a distributed operating system is more difficult than designing a centralized operating system for several reasons.
- The reasons are
    - Transparency
    - Reliability
    - Flexibility
    - Performance
    - Scalability
    - Heterogeneity
    - Security
    - Emulation of Existing Operating Systems

Transparency

- The main goals of a distributed operating system are to make the existence of multiple computers invisible (transparent) and provide a single system image to its users.
- That is, a distributed operating system must be designed in such a way that a collection of distinct machines connected by a communication subsystem appears to its users as a virtual uniprocessor.
- Achieving complete transparency is a difficult task and requires that several different aspects of transparency be supported by the distributed operating system.

- The eight forms of transparency identified by the International Standards Organization's Reference Model for Open Distributed Processing [ISO 1992] are
    - Access Transparency
    - Location Transparency
    - Replication Transparency
    - Failure Transparency
    - Migration Transparency
    - Concurrency Transparency
    - Performance Transparency
    - Scaling Transparency.

Access Transparency:

- Access transparency means that users should not need or be able to recognize whether a resource (hardware or software) is remote or local. This implies that the distributed operating system should allow users to access remote resources in the same way as local resources.

Location Transparency

- The two main aspects of location transparency are as follows:
    1. **Name transparency.** This refers to the fact that the name of a resource (hardware or software) should not reveal any hint as to the physical location of the resource.
    2. **User mobility.** This refers to the fact that no matter which machine a user is logged onto, he or she should be able to access a resource with the same name.

Replication Transparency

- For better performance and reliability, almost all distributed operating systems have the provision to create replicas (additional copies) of files and other resources on different nodes of the distributed system. In these systems, both the existence of multiple copies of a replicated resource and the replication activity should be transparent to the users.

Failure Transparency

- Failure transparency deals with masking from the users' partial failures in the system, such as a communication link failure, a machine failure, or a storage device crash. A distributed operating system having failure transparency property will continue to function, perhaps in a degraded form, in the face of partial failures.

Migration Transparency

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

Concurrency Transparency

- Concurrency transparency means that each user has a feeling that he or she is the sole user of the system and other users do not exist in the system.
- For providing concurrency transparency, the following properties should be ensured
    - Event ordering property
    - Mutual exclusion property
    - No starvation property
    - No deadlock property

Performance Transparency

- The aim of performance transparency is to allow the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system.

Scaling Transparency

- The aim of scaling transparency is to allow the system to expand in scale without disrupting the activities of the users.

Reliability

- In general, distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources.
- However, the existence of multiple instances of the resources alone cannot increase the system's reliability. Rather, the distributed operating system, which manages these resources, must be designed properly to increase the system's reliability by taking full advantage of this characteristic feature of a distributed system.
- A fault is a mechanical or algorithmic defect that may generate an error. A fault in a system causes system failure. Depending on the manner in which a failed system behaves, system failures are of two types: fail-stop failure and Byzantine failure.

- For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and recover from faults. Commonly used methods for dealing with these issues are fault avoidance and fault tolerance.
- **Fault Avoidance:** Deals with designing the component of the system in such a way that the occurrence of faults is minimized.
- **Fault Tolerance:** Ability of a system to continue functioning in the event of partial system failure. The following facts are used to improve fault tolerance ability
    - Redundancy techniques
    - Distributed control
- Fault Detection and Recovery: Deals with the use of hardware and software mechanisms to determine the occurrence of the failure and then to correct the system to a state acceptable for continued operation. The commonly used techniques are
    - Atomic Transactions
    - Stateless Servers
    - Acknowledgements and timeout-based retransmissions of messages.

Flexibility

- Another important issue in the design of distributed operating systems is flexibility.
- Flexibility is the most important feature for open distributed systems. The design of a distributed operating system should be flexible due to the following reasons:

  1. Ease of modification.
  2. Ease of enhancement

Performance

- If a distributed system is to be used, its performance must be at least as good as a centralized system.
- That is, when a particular application is run on a distributed system, its overall performance should be better than or at least equal to that of running the same application on a single-processor system.
- To achieve this goal, it is important that the various components of the operating system of a distributed system be designed properly; otherwise, the overall performance of the distributed system may turn out to be worse than a centralized system. Some design principles considered useful for better performance are as follows:

    1. Batch if possible.

2. Cache whenever possible.
3. Minimize copying of data.
4. Minimize network traffic.
5. Take advantage of fine-grain parallelism for multiprocessing.

Scalability

- Scalability refers to the capability of a system to adapt to increased service load.
- It is inevitable that a distributed system will grow with time since it is very common to add new machines or an entire sub network to the system to take care of increased workload or organizational changes in a company.
- Therefore, a distributed operating system should be designed to easily cope with the growth of nodes and users in the system.
- That is, such growth should not cause serious disruption of service or significant loss of performance to users. Some guiding principles for designing scalable distributed systems are as follows:
    1. Avoid centralized entities.
    2. Avoid centralized algorithms.
    3. Perform most operations on client workstations.

Heterogeneity

- A heterogeneous distributed system consists of interconnected sets of dissimilar hardware or software systems.
- Because of the diversity, designing heterogeneous distributed systems is far more difficult than designing homogeneous distributed systems in which each system is based on the same, or closely related, hardware and software.
- However, as a consequence of large scale, heterogeneity is often inevitable in distributed systems. Furthermore, often heterogeneity is preferred by many users because heterogeneous distributed systems provide the flexibility to their users of different computer platforms for different applications.

Security

- In order that the users can trust the system and rely on it, the various resources of a computer system must be protected against destruction and unauthorized access.
- Enforcing security in a distributed system is more difficult than in a centralized system because of the lack of a single point of control and the use of insecure networks for data

communication. Therefore, as compared to a centralized system, enforcement of security in a distributed system has the following additional requirements:

1. It should be possible for the sender of a message to know that the message was received by the intended receiver.
2. It should be possible for the receiver of a message to know that the message was sent by the genuine sender.
3. It should be possible for both the sender and receiver of a message to be guaranteed that the contents of the message were not changed while it was in transfer.

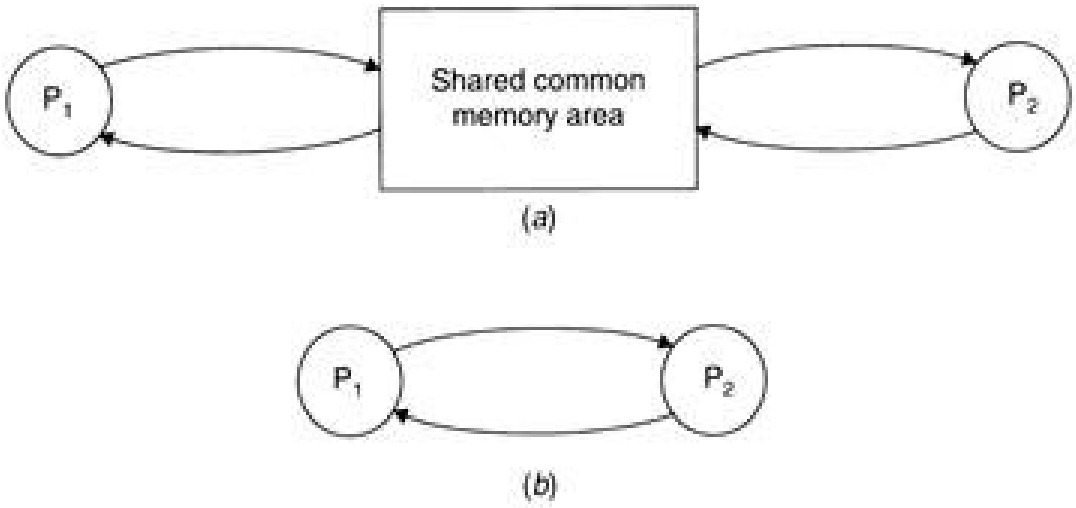
- Cryptography is the only known practical method for dealing with these security aspects of a distributed system.

Emulation of Existing Operating System

- For commercial success, it is important that a newly designed distributed operating system be able to emulate existing popular operating systems such as UNIX.
- With this property, new software can be written using the system call interface of the new operating system to take full advantage of its special features of distribution, but a vast amount of already existing old software can also be run on the same system without the need to rewrite them.
- Therefore, moving to the new distributed operating system will allow both types of software to be run side by side.

MESSAGE PASSING

- A process is a program in execution.
- If two computers of a distributed system are communicating with each other, the two process running on each computer are in communication with each other.
- In distributed systems, to achieve some common goal, processes executing on different computers often need to communicate with each other.
- Therefore a distributed operating system needs to provide Inter Process Communication (IPC) mechanisms to facilitate such communication activities.
- Inter Process Communication requires information sharing among two or more processes.
- Two basic methods for information sharing are
    - Original Sharing or shared data approach
    - Copy Sharing or message passing approach



- In the shared-data approach, the information to be shared is placed in a common memory area that is accessible to all processes involved in an IPC.
- In the message-passing approach, the information to be shared is physically copied from the sender process's space to the address space of all the receiver processes, and this is done by transmitting the data to be copied in the form of messages (message is a block of information).
- A message-passing system is a subsystem of distributed operating system that provides a set of message-based IPC protocols.
- It enables processes to communicate by exchanging messages and allows programs to be written by using simple communication primitives, such as send and receive.
- It serves as a suitable infrastructure for building other higher level IPC systems such as Remote Procedure Call (RPC)

DESIRABLE FEATURES OF A GOOD MESSAGE-PASSING SYSTEM

Simplicity

- A message passing system should be simple and easy to use. It should be possible to communicate with old and new applications, with different modules without the need to worry about the system and network aspects.

Uniform Semantics

- In a distributed system, a message-passing system may be used for the following two types of Inter Process Communication:
    - Local Communication, in which the communicating processes are on the same node;
    - Remote Communication, in which the communicating processes are on different nodes.

- Semantics of remote communication should be as close as possible to those of local communications. This is an important requirement for ensuring that the message passing is easy to use.

Efficiency

- An IPC protocol of a message-passing system can be made efficient by reducing the number of message exchanges, as far as practicable, during the communication process. Some optimizations normally adopted for efficiency include the following:
    - Avoiding the costs of establishing and terminating connections between the same pair of processes for each and every message exchange between them;
    - Minimizing the costs of maintaining the connections;
    - Piggybacking of acknowledgement of previous messages with the next message during a connection between a sender and a receiver that involves several message exchanges.

Correctness

- Correctness is a feature related to IPC protocols for group communication. Issues related to correctness are as follows:
    - Atomicity;
    - Ordered delivery;
    - Survivability.
- Atomicity ensures that every message sent to a group of receivers will be delivered to either all of them or none of them.
- Ordered delivery ensures that messages arrive to all receivers in an order acceptable to the application.
- Survivability guarantees that messages will be correctly delivered despite partial failures of processes, machines, or communication links.

Reliability

- A reliable IPC protocol can cope with failure problems and guarantees the delivery of a message.
- A good message passing system must have IPC protocols for the support the following reliability features:
    - Lost message handling – involves acknowledgments and retransmission on the basis of timeouts.

- Duplicate message handling – involves generating and assigning appropriate sequence numbers to messages

Flexibility

- IPC protocols of a message passing system must be flexible enough to cater to the various needs of different applications.
- The users have the flexibility to choose and specify the types and levels of reliability and correctness requirements of their applications.
- IPC primitives must also have the flexibility to permit any kind of control flow between the cooperating processes, including synchronous and asynchronous send / receive.

Security

- A good message passing system must also be capable of providing a secure end-to-end communication.
- The steps necessary for secure communication include the following:
  - Authentication of the receivers of the message by the sender.
  - Authentication of the sender of a message by its receivers.
  - The encryption of a message before sending it over the network.

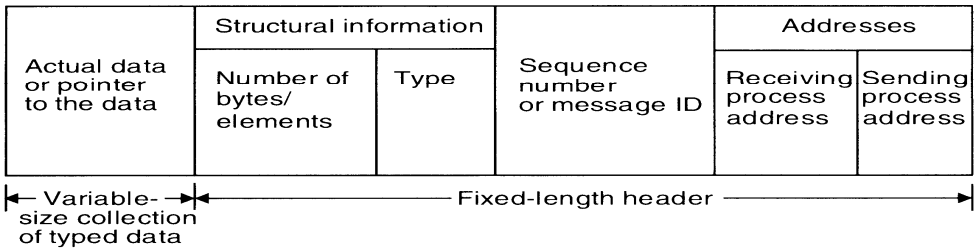
Portability

- There are two different aspects of portability in a message passing system:
  - The Message passing system should itself be portable
  - The application written by using primitives of the IPC protocol should be portable.

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.
  - **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.



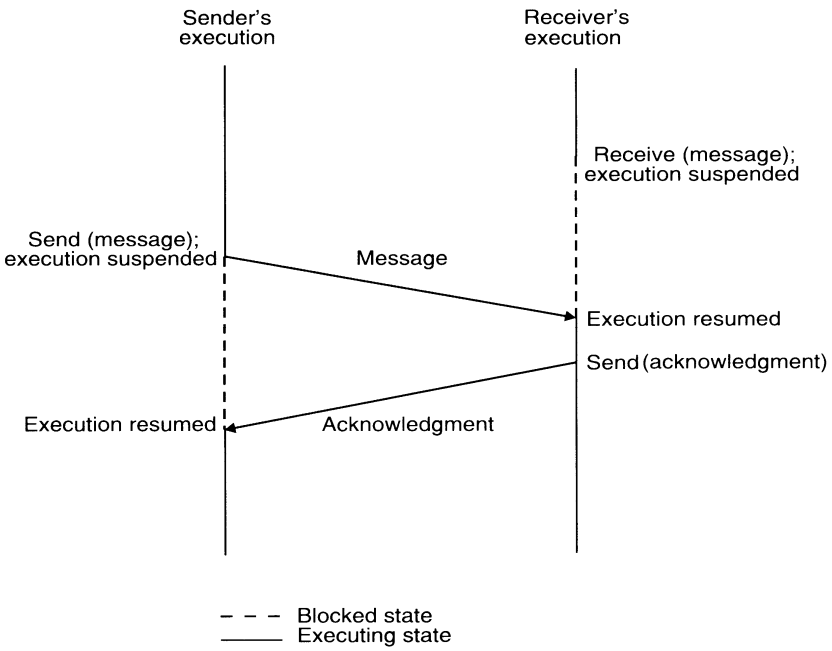
A Typical Message Structure

- In a message oriented IPC protocol, the sending process determines the actual contents of a message and the receiving process is aware of how to interpret the contents.
- Special primitives are explicitly used for sending and receiving the messages. Therefore the users are fully aware of the message formats used in the communication process and the mechanisms used to send and receive messages.
- The following are the issues need to be considered in the design of an IPC protocol for a a message passing system:
    - Who is the sender?
    - Who is the receiver?
    - Is there one receiver or many receivers?
    - Is the message guaranteed to have been accepted by its receivers?
    - Does the sender need to wait for a reply?
    - What should be done if a catastrophic event such as a node crash or a communication link failure occurs during the course of communication?
    - What should be done if the receiver is not ready to accept the message: Will the message be discarded or stored in the buffer? In the case of buffering, what should be done if the buffer is full?
    - If there are several outstanding messages for a receiver, can it choose the order in which to service the outstanding messages?

SYNCHRONIZATION

- A central issue in the communication structure is the synchronization imposed on the communicating processes by the communication primitives.
- The semantics used for synchronization may be broadly classified as **blocking and nonblocking** types.

- A primitive is said to have nonblocking semantics if its invocation does not block the execution of its invoker (the control returns almost immediately to the invoker); otherwise a primitive is said to be of the blocking type.
- In case of a **blocking send primitive**, after execution of the send statement, the sending process is blocked until it receives an acknowledgement from the receiver that the message has been received. On the other hand, for **nonblocking 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.
- In the case of **blocking receive primitive**, after execution of the receive statement, the receiving process is blocked until it receives a message. On the other hand, for a **nonblocking 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 nonblocking receives 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:
    - **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.
    - **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.
- A variant of the nonblocking receives primitive is the **conditional receive primitive**, which also returns control to the invoking process almost immediately, either with a message or with an indicator that no message is available.
- 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.
- The main drawback of synchronous communication is that it limits concurrency and communication, is subject to deadlocks.



BUFFERING

- In the standard message passing model, messages can be copied many times: from the user buffer to the kernel buffer (the output buffer of a channel), from the kernel buffer of the sending computer (process) to the kernel buffer in the receiving computer (the input buffer of a channel), and finally from the kernel buffer of the receiving computer (process) to a user buffer.

Null Buffer (No Buffering)

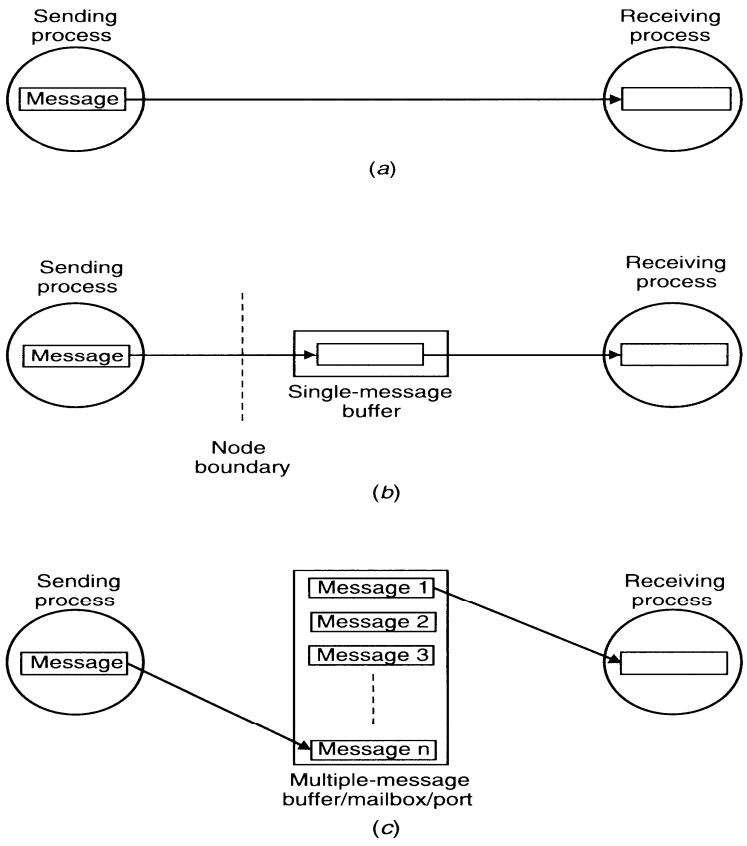
- In this case, there is no place to temporarily store the message. Hence, one of the following implementation strategies may be used:
    - The message remains in the sender process's address space and the execution of the send is delayed until the receiver executes the corresponding receive.
    - The message is simply discarded and the time-out mechanism is used to resend the message after a timeout period. The sender may have to try several times before succeeding.
- The three types of buffering strategies used in interprocess communication are explained below with its diagram

Single-Message Buffer

- In single-message buffer strategy, a buffer having a capacity to store a single message is used on the receiver's node. This strategy is usually used for synchronous communication, an application module may have at most one message outstanding at a time.

Unbounded-Capacity Buffer

- In the asynchronous mode of communication, since a sender does not wait for the receiver to be ready, there may be several pending messages that have not yet been accepted by the receiver. Therefore, an unbounded-capacity message-buffer that can store all unreceived messages is needed to support asynchronous communication with the assurance that all the messages sent to the receiver will be delivered.



Finite-Bound Buffer

- The unbounded capacity of a buffer is practically impossible. Therefore, in practice, systems using asynchronous mode of communication use finite-bound buffers, also known as multiple-message buffers. In this case message is first copied from the sending process's memory into the receiving process's mailbox and then copied from the mailbox to the receiver's memory when the receiver calls for the message.
- When the buffer has finite bounds, a strategy is also needed for handling the problem of a possible buffer overflow. The buffer overflow problem can be dealt with in one of the following two ways:
    - **Unsuccessful communication:** In this method, message transfers, simply fail, whenever there is no more buffer space and an error is returned.

- **Flow-controlled communication:** The second method is to use flow control, which means that the sender is blocked until the receiver accepts some messages, thus creating space in the buffer for new messages. This method introduces a synchronization between the sender and the receiver and may result in unexpected deadlocks. Moreover, due to the synchronization imposed, the asynchronous send does not operate in the truly asynchronous mode for all send commands.

MULTIDATAGRAM MESSAGES

- Almost all networks have an upper bound of data that can be transmitted at a time. This size is known as Maximum Transfer Unit (MTU). A message whose size is greater than MTU has to be fragmented into multiples of the MTU, and then each fragment has to be sent separately. Each packet is known as a datagram. Messages larger than the MTU are sent in miltipackets, and are known as multidatagram messages. Messages smaller than the MTU are known as single datagram messages.

ENCODING AND DECODING OF MESSAGES

- A message data should be meaningful to the receiving process. This implies that, ideally, the structure of program objects should be preserved while they are being transmitted from the address space of the sending process to the address space of the receiving process. However, even in homogeneous systems, it is very difficult to achieve this goal mainly because of two reasons:
  - An absolute pointer value loses its meaning when transferred from one process address space to another.
  - Different program objects occupy varying amount of storage space. To be meaningful, a message must normally contain several types of program objects, such as long integers, short integers, variable-length character strings, and so on.
- In transferring program objects in their original form, they are first converted to a stream form that is suitable for transmission and placed into a message buffer. This conversion process takes place on the sender side and is known as encoding of message data.
- The encoded message, when received by the receiver, must be converted back from the stream form to the original form before it can be used. The process of reconstruction of program object from the message data on the receiver side is known as decoding of message data.
- One of the following two representations may by used for the encoding and decoding of a message data:

- In tagged representation the type of each program object along with its value is encoded in the message.
- In untagged representation the message data only contain program object. No information is included in the message data to specify the type of each program object.

PROCESS ADDRESSING

- Another important issue in message-based communication is addressing (or naming) of the parties involved in an interaction. For greater flexibility a message-passing system usually supports two types of process addressing:
  - **Explicit addressing:** The process with which communication is desired is explicitly named as a parameter in the communication primitive used.
  - **Implicit addressing:** The process willing to communicate does not explicitly name a process for communication (the sender names a server instead of a process). This type of process addressing is also known as functional addressing.

(a) **send** (process_id, message)
Send a message to the process identified by "process_id".

(b) **receive** (process_id, message)
Receive a message from the process identified by "process_id".

(c) **send_any** (service_id, message)
Send a message to any process that provides the service of type "service_id".

(d) **receive_any** (process_id, message)
Receive a message from any process and return the process identifier ("process_id") of the process from which the message was received.

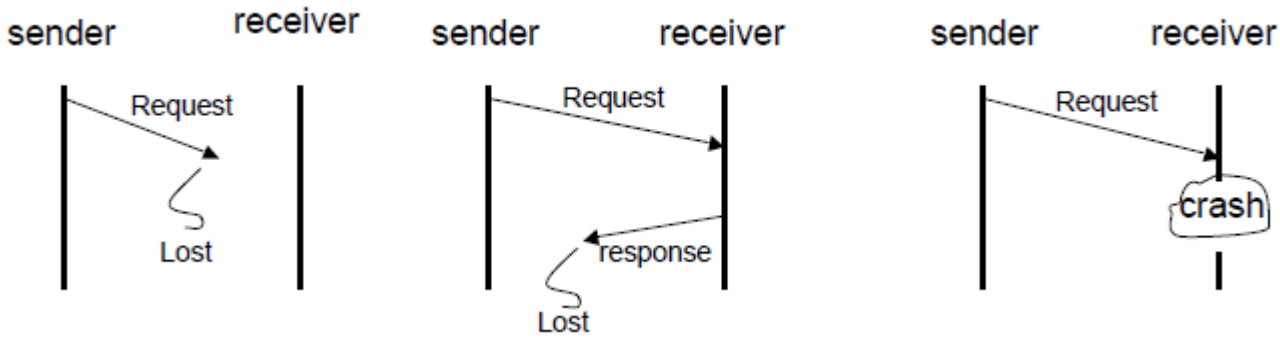
Methods to Identify a Process (naming)

- A simple method to identify a process is by a combination of machine_id and local_id. The local_id part is a process identifier, or a port identifier of a receiving process, or something else that can be used to uniquely identify a process on a machine. The machine_id part of the address is used by the sending machine's kernel to send the message to the receiving process's machine, and the local_id part of the address is then used by the kernel of the receiving process's machine to forward the message to the process for which it is intended.
- A drawback of this method is that it does not allow a process to migrate from one machine to another if such a need arises.

- To overcome the limitation of the above method, process can be identified by a combination of the following three fields: machine_id, local_id and machine_id.
    - The first field identifies the node on which the process was created.
    - The second field is the local identifier generated by the node on which the process was created.
    - The third field identifies the last known location (node) of the process.
- Another method to achieve the goal of location transparency in the process addressing is to use a two-level naming scheme for processes. In this method each process has two identifiers: a high-level name that is machine independent (an ASCII string) and the low-level name that is machine dependent (such as pair (machine_id, local_id). A name server is used to maintain a mapping table that maps high-level names of processes to their low-level names.

FAILURE HANDLING

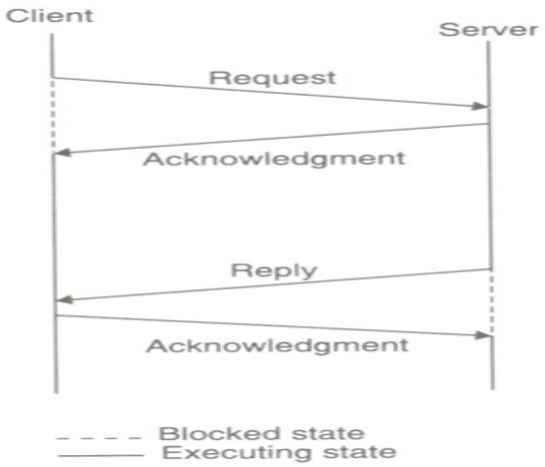
- During interprocess communication, partial failures such as a node crash or communication link failure may lead to the following problems:
    - **Loss of request message:** This may happen either due to the failure of communication link between the sender and receiver or because the receiver's node is down at the time the request message reaches there.
    - **Loss of response message:** This may happen either due to the failure of communication link between the sender and receiver or because the sender's node is down at the time the response message reaches there.
    - **Unsuccessful execution of the request:** This may happen due to the receiver's node crashing while the request is being processed.



Possible problems in IPC due to different types of system failures

- Four-message reliable IPC protocol for client-server communication between two processes works as follows:

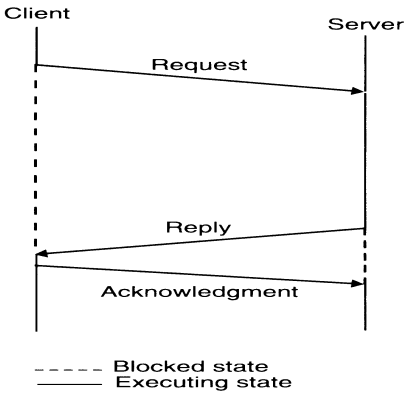
- The client sends a request message to the server.
- When the request message is received at the server's machine, the kernel of that machine returns an acknowledgment message to the kernel of the client machine. If the acknowledgment is not received within the timeout period, the kernel of the client machine retransmits the request message.
- When the server finishes processing the client's request, it returns a reply message (containing the result of processing) to the client.
- When the reply is received at the client machine, the kernel of that machine returns an acknowledgment message to the kernel of the server machine. If the acknowledgment message is not received within the timeout period, the kernel of the server machine retransmits the reply message.



The four message reliable IPC

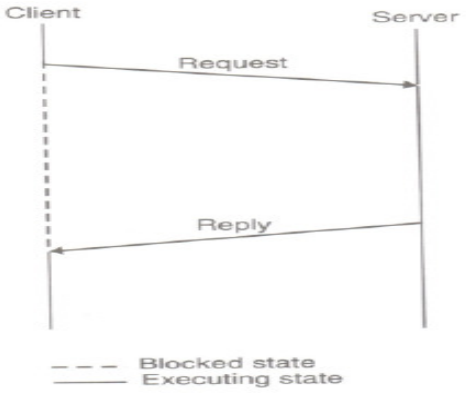
- In client-server communication, the result of the processed request is sufficient acknowledgment that the request message was received by the server. Based on this idea, a three-message reliable IPC protocol for client-server communication between two processes works as follows:
  - The client sends a request message to the server.
  - When the server finishes processing the client's request, it returns a reply message (containing the result of processing) to the client. The client remains blocked until the reply is received. If the reply is not received within the timeout period, the kernel of the client machine retransmits the request message.

- When the reply message is received at the client's machine, the kernel of that machine returns an acknowledgment message to the kernel of the server machine. If the acknowledgment message is not received within the timeout period, the kernel of the server machine retransmits the reply message.



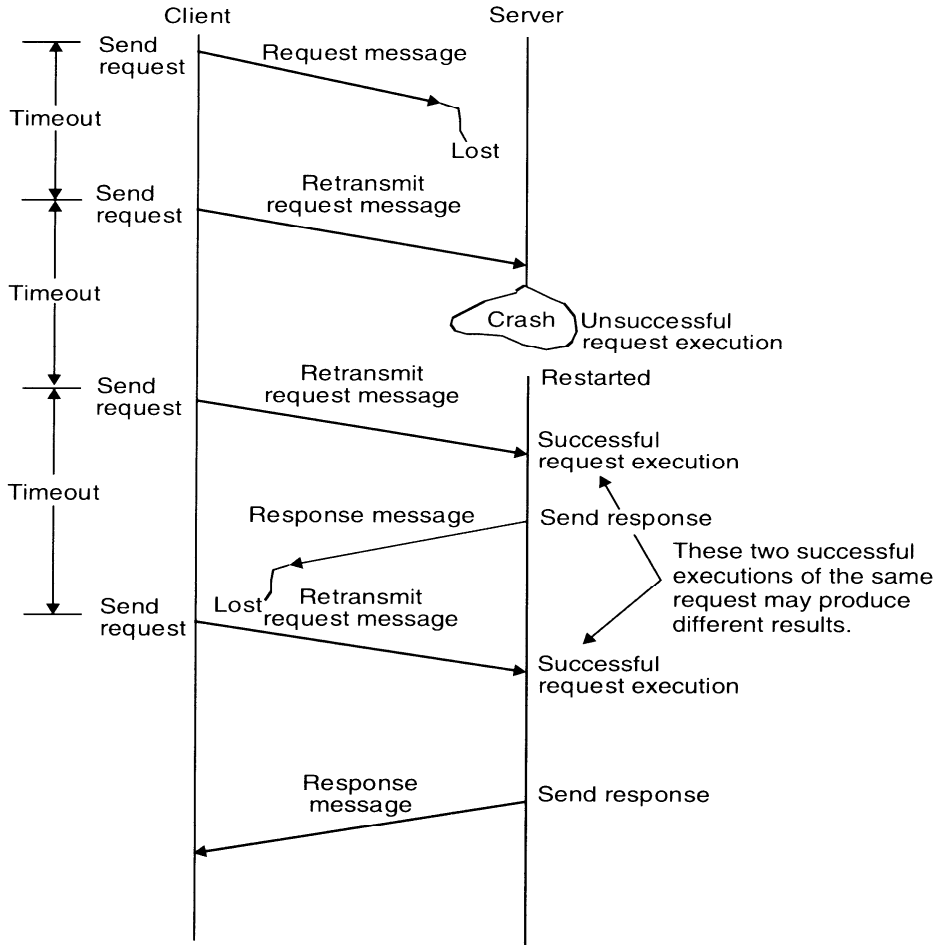
The three message reliable IPC

- A problem occurs if a request processing takes a long time. If the request message is lost, it will be retransmitted only after the timeout period, which has been set to a large value to avoid unnecessary retransmission of the request message. On the other hand, if the timeout value is not set to properly take into consideration the long time needed for request processing, unnecessary retransmissions of the request message will take place.
- The following protocol may be used to handle this problem:
  - The client sends a request message to the server.
  - When the request message is received at the server's machine, the kernel of that machine starts a timer. If the server finishes processing the client's requests and returns the reply message to the client before the timer expires, the reply serves as the acknowledgment of the request message. Otherwise, a separate acknowledgment is sent by the kernel of the server machine to acknowledge the request message. If an acknowledgement is not received within the timeout period, the kernel of the client machine retransmits the request message.
  - When the reply message is received, at the client's machine, the kernel of that machine returns an acknowledgment message to the kernel of the server machine. If the acknowledgment message is not received within the timeout period, the kernel of the server retransmits the reply message.
- A message-passing system may be designed to use the following two-message IPC protocol for client-server communication between two processes:



The two message reliable IPC

- The client sends a request message to the server and remains blocked until a reply is received from the server.
- When the server finishes processing the client's request, it returns a reply message (containing the result of processing) to the client. If the reply is not received within the timeout period, the kernel of the client machine retransmits the request message.

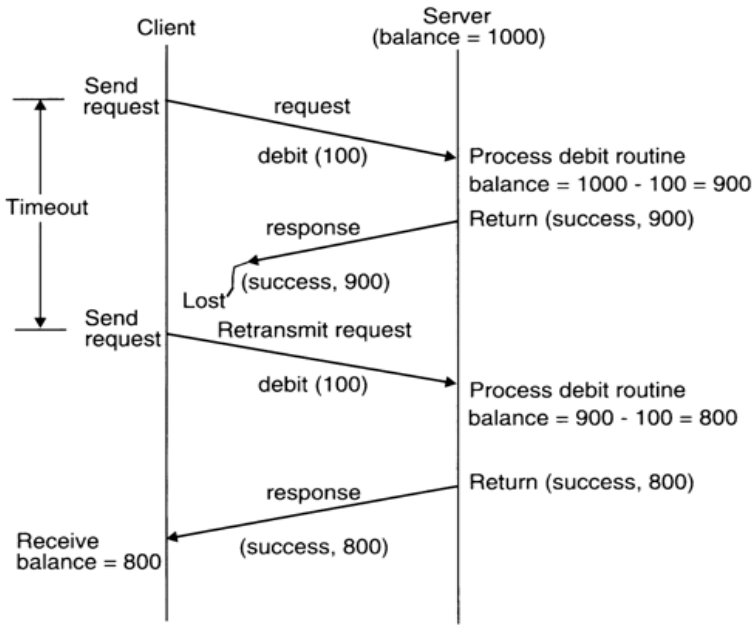


An example of fault tolerant communication between a client and a server.

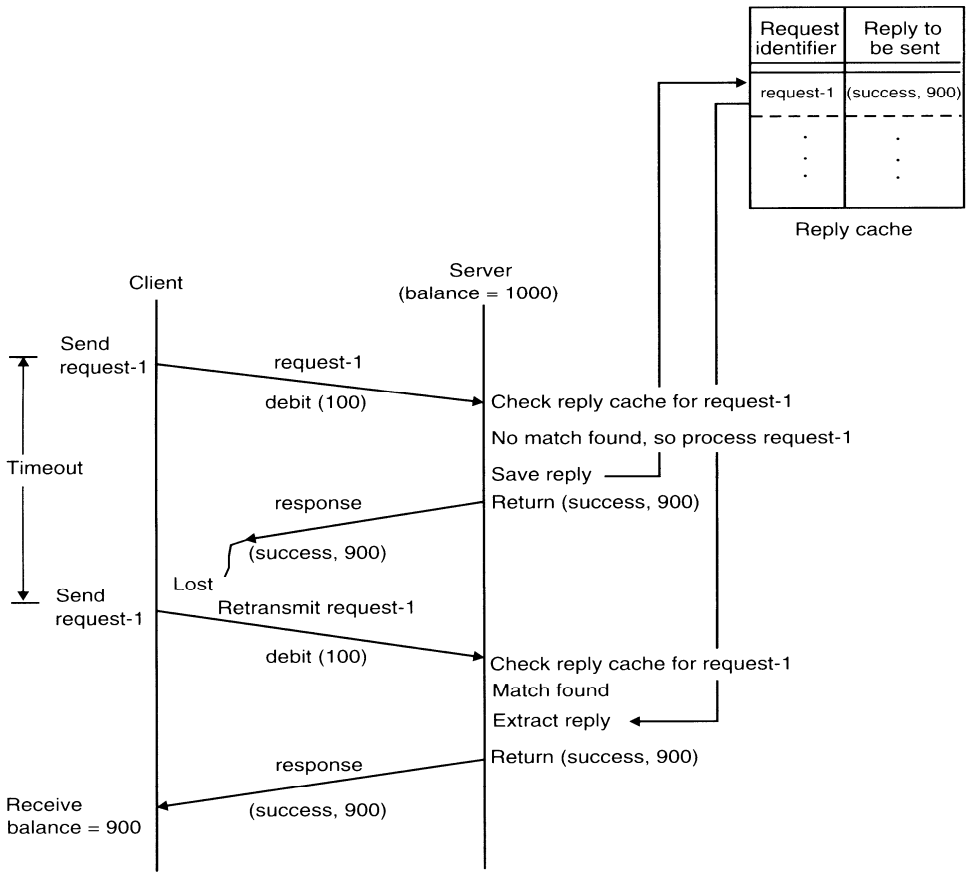
Idempotency

- Idempotency basically means "repeatability". That is, an idempotent operation produces the same results without any side effects, no matter how many times it is performed with the same arguments.

```
debit (amount)
{
    if (balance >= amount)
    {   balance = balance - amount;
        return ("success", balance);
    }
    else return ("failure", balance);
}
```



A Nonidempotent Routine

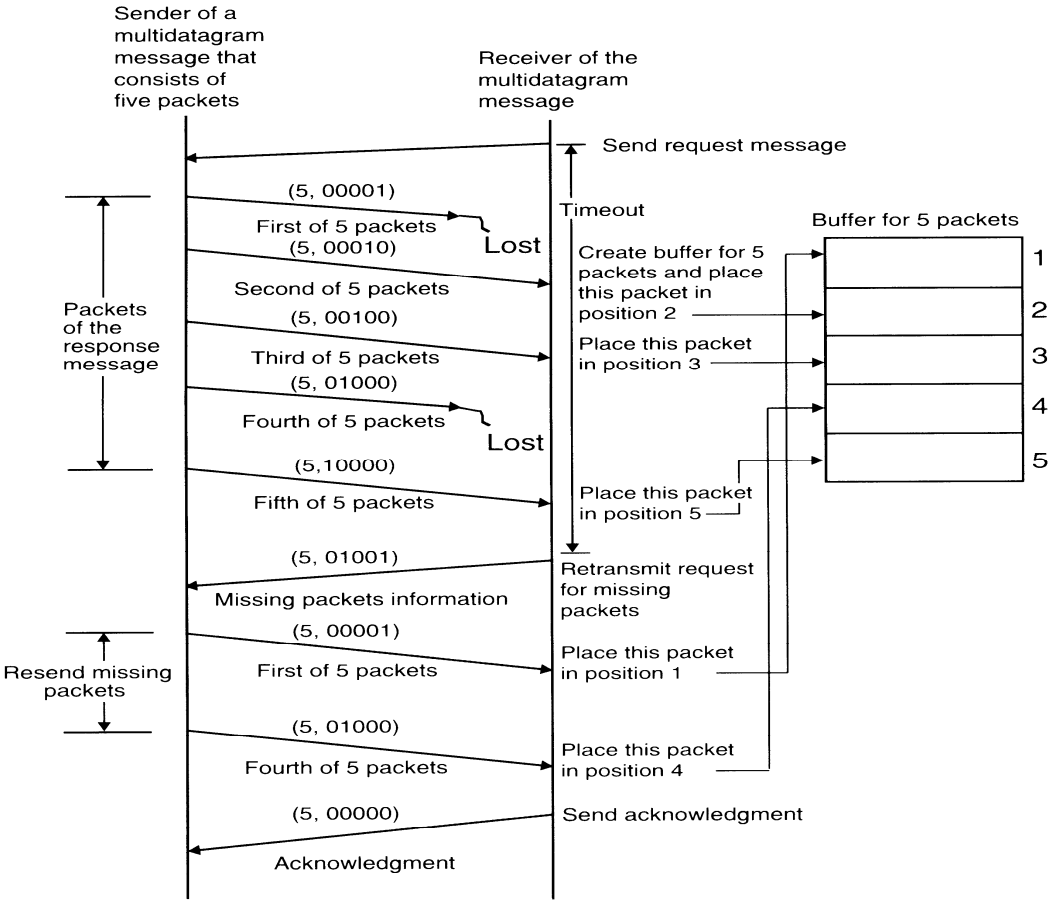


An example of exactly-once semantics using request identifiers and reply cache

Keeping track of Lost and Out-of-Sequence Packet in Multidatagram Messages

- A message transmission can be considered to be complete only when all the packets of the message have been received by the process to which it is sent. For successful completion of a multidatagram message transfer, reliable delivery of every packet is important. A simple way to ensure this is to acknowledge each packet, separately (called stop-and-wait protocol). To improve communication performance, a better approach is to use a single acknowledgment packet for all the packets of a multidatagram message (called blasts protocol). However, when this approach is used, a node crash or a communication link failure may lead to the following problems:
    - One or more packets of the multidatagram message are lost in communication.
    - The packets are received out of sequence by the receiver.

- An efficient mechanism to cope with these problems is to use a bitmap to identify the packets of a message.



GROUP COMMUNICATION

- The most elementary form of message-based interaction is one-to-one communication (also known as point-to-point, or unicast communication) in which a single-sender process sends a message to a single-receiver process. For performance and ease of programming, several highly parallel distributed applications require that a message-passing system should also provide the group communication facility. Depending on a single or multiple senders and receivers, the following three types of group communication are possible:
    - One to many (single sender and multiple receivers).
    - Many to one (multiple senders and single receivers).
    - Many to many (multiple senders and multiple receivers).

One-to-Many Communication

- In this scheme, there are multiple receivers for a message sent by a single sender. One-to-many scheme is also known as multicast communication. A special case of multicast communication

is broadcast communication, in which the message is sent to all processors connected to a network.

- **Group Management**
    - In case of one-to-many communication, receiver processes of a message form a group. Such groups are of two types – closed and open. A closed group is one in which only the members of the group can send a message to the group. An outside process cannot send a message to the group as a whole, although it may send a message to an individual member of the group. On the other hand, an open group is one in which any process in the system can send a message to the group as a whole.
- **Group Addressing**
    - A two-level naming scheme is normally used for group addressing. The high-level group name is an ASCII string that is independent of the location of the processes in the group. On the other hand, the low-level group name depends to a large extent on the underlying hardware.
    - On some networks it is possible to create a special network address to which multiple machines can listen. Such a network address is called a multicast address. Therefore, in such systems a multicast is used as a low-level name for a group.
    - Some networks that do not have the facility to create multicast address may have broadcast facility. A packet sent to a broadcast address is automatically delivered to all machines in the network. In this case, the software of each machine must check to see if the packet is intended for it.
    - If a network does not support either the facility to create multicast address or the broadcasting facility, a one-to-one communication mechanism has to be used to implement the group communication facility. That is, the kernel of the sending machine sends the message packet, separately to each machine that has a process belonging to the group. Therefore, in this case, the low-level name of a group contains a list of machine identifiers of all machines that have a process belonging to the group.
- **Buffered and Unbuffered Multicast**
    - Multicasting is an asynchronous communication mechanism. This is because multicast sends cannot be synchronous due to the following reasons:
    - It is unrealistic to expect a sending process to wait until all the receiving processes that belong to the multicast group are ready to receive the multicast message.

- The sending process may not be aware of all the receiving processes that belong to the multicast group.
- For an unbuffered multicast, the message is not buffered for the receiving process and is lost if the receiving process is not in a state ready to receive it. Therefore, the message is received only by those processes of the multicast group that are ready to receive it. On the other hand, for a buffered multicast, the message is buffered for the receiving process, so each process of the multicast group will eventually receive the message.

- **Send-to-All and Bulletin-Board Semantics**
  - Ahamad and Berstain [1985] described the following two types of semantics for one-to-many communications:
  - **Send-to-all semantics:** A copy of the message is sent to each process of the multicast group and message is buffered until it is accepted by the process.
  - **Bulletin-board semantics:** A message to be multicast is addressed to a channel instead of being sent to every individual process of the multicast group. From a logical point of view, the channel plays the role of a bulletin board. A receive process copies the message from the channel instead of removing it when it makes a receive request on the channel.
  - Bulletin-board semantics is more flexible than send-to-all semantics because it takes care of the following two factors that are ignored by send-to-all semantics:
    - The relevance of a message to a particular receiver may depend on the receiver's state.
    - Messages not accepted within a certain time after transmission may no longer be useful; their value may depend on the sender's state.
- **Flexible Reliability in Multicast Communication**
  - Different applications require different degrees of reliability. The sender of a multicast message can specify the number of receivers from which a response message is expected. In one-to-many communication, the degree of reliability is normally expressed in the following forms:
    - The 0-reliability. No response is expected by the sender from any of the receivers.
    - The 1-reliability. The sender expects a response from any of the receivers.

        - The m-out-of-n-reliable. The multicast group consists of n receivers and the sender expect a response from m (1<m<n) of the receivers.
        - All-reliable. The sender expects a response message from all the receivers of the multicast group.

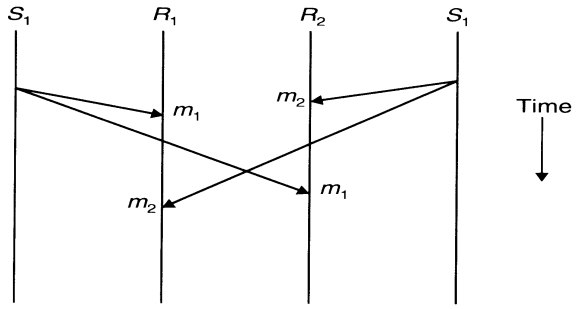
- **Atomic Multicast**
    - Atomic multicast (reliable multicast) has an all-or-nothing property. That is, when a message is sent to a group by atomic multicast, it is either received by all the surviving (correct) processes that are members of the group or else it is not received by any of them.
    - When a sender is reliable and network partition is excluded, a simple way to implement atomic multicast is to multicast a message, with the degree of reliability requirement being all-reliable. In this case, the kernel of the sending machine sends the message to all members of the group and waits for an acknowledgment from each member.
    - The above approach is not sufficient when considering possible failures of the sender's machine or a receiver's machine.
    - One method to implement atomic (reliable) multicast is the following.
    - Each message has a message identifier field to distinguish it from all other messages and a field to indicate that it is an atomic multicast message. The sender sends the message to a multicast group. The kernel of the sending machine sends the message to all members of the group and uses timeout-based retransmissions as in the previous method. A process that receives the message checks its message identifier field to see if it is a new message. If not, it is simply discarded. Otherwise, the receiver checks to see if it is an atomic multicast message. If so, the receiver also performs an atomic multicast of the same message, sending it to the same multicast group. The kernel of this machine treats this message as an ordinary atomic multicast message and uses timeout-based retransmission when needed. In this way, each receiver of an atomic multicast message will perform an atomic multicast of the message to the same multicast group.
    - The method ensures that eventually all the surviving processes of the multicast group will receive the message even if the sender machine fails after sending the message or a receiver machine fails after receiving the message.

Many-to-One Communication

- In this scheme, multiple senders send messages to a single receiver. The single receiver may be selective or nonselective.
- A selective receiver specifies a unique sender; a message exchange takes place only if that sender sends a message. On the other hand, a nonselective receiver specifies a set of senders, and if any one sender in the set sends a message to this receiver, a message exchange takes place.
- An important issue related to the many-to-one communication scheme is nondeterminism. It is not known in advance which member (or members) of the group will have its information available first.

Many-to-Many Communication

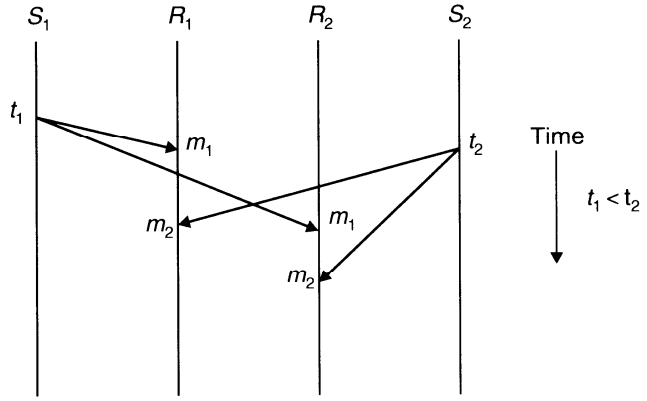
- In this scheme, multiple senders send messages to multiple receivers.
- An important issue related to many-to-many communication scheme is that of ordered message delivery. Ordered message delivery ensures, that all messages are delivered to all receivers in an order acceptable to the application.



No ordering constraints for message delivery

- **Absolute Ordering**
    - The absolute ordering semantics ensures that all messages are delivered to all receiver processes in the exact order in which they were sent.
    - One method to implement these semantics is to use global timestamps as message identifiers.
    - That is, the system is assumed to have a clock at each machine and all clocks are synchronized with each other, and when a sender sends a message, the clock value (timestamp) is taken as the identifier of that message and the timestamp is embedded in the message.

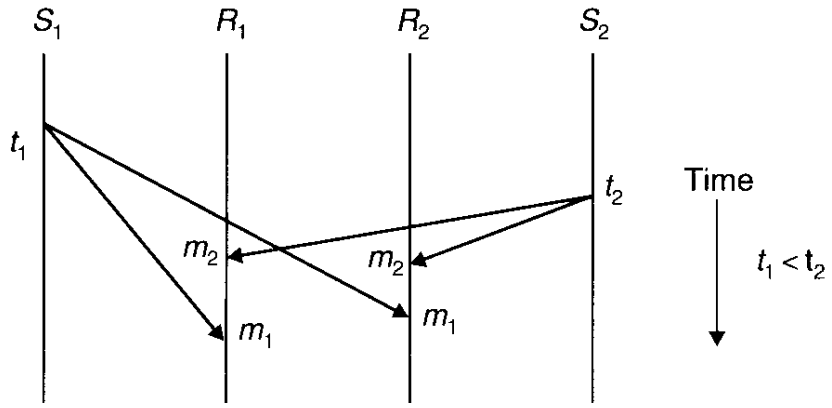
- To implement absolute ordering semantics, the kernel of each receiver's machine saves all incoming messages meant for a receiver in a separate queue.



Absolute ordering of messages

- A sliding-window mechanism is used to periodically deliver the message from the queue to the receiver. That is, a fixed time interval is selected as the window size, and periodically all messages whose timestamp values fall within the current window are delivered to the receiver. Messages whose timestamp values fall outside the window are left in the queue because of the possibility that a tardy message having a timestamp value lower than that of any of the messages in the queue might still arrive.
- The window size must be properly chosen taking into consideration the maximum possible time that may be required by a message to go from one machine to any other machine in the network.

- **Consistent Ordering (Total Ordering)**
  - Absolute-ordering semantics requires globally synchronized clocks, which are not easy to implement. Moreover, absolute ordering is not really what many applications need to function correctly.



Consistent ordering of messages

- Therefore, instead of supporting absolute ordering semantics, most systems support consistent-ordering semantics (total order semantics). This semantics ensures that all messages are delivered to all receiver processes in the same order. However, this order may be different from the order in which messages were sent.
- One method to implement consistent-ordering semantics is to make the many-to-many scheme appear as a combination of many-to-one and one-to-many schemes. That is, the kernels of the sending machines send messages to a single receiver (known as sequencer) that assigns a sequence number to each message and then multicasts it. The kernel of each receiver's machine saves all incoming messages meant for a receiver in a separate queue.
- The sequencer-based method for implementing consistent-ordering semantics are subject to asingle point of failure and hence has poor reliability. A distributed algorithm for implementing consistent-ordering semantics that does not suffer from this problem is the ABCAST protocol of the ISIS system. It assigns a sequence number to a message by distributed agreement among the group members and the sender, and works as follows:

1. The sender assigns a temporary sequence number (integer) to the message and sends it to all members of the multicast group. The sequence number assigned by the sender must be larger than any previous sequence number used by the sender. Therefore, a simple counter can be used by the sender to assign sequence numbers to its messages.
2. On receiving the message, each member of the group returns a proposed sequence number to the sender. A member (i) calculates its proposed sequence number by using the function: Max (Fmax, Pmax) + 1 + i/N, where integer Fmax is truncated the largest final sequence number agreed upon so far for a message received by the group (each member makes a record of this when a final sequence number is agreed upon), integer Pmax is truncated the largest proposed sequence number by this member, and N is the total number of members in the multicast group.
3. When the sender receives the proposed sequence number from all the members, it selects the largest one as the final sequence number for the message and sends it to all members in a commit message. The chosen final sequence number is guaranteed to be unique because of the term i/N in the function used for the calculation of a proposed sequence number.

4. On receiving the commit message, each member attaches the final sequence number to the message.
5. Committed messages with final sequence numbers are delivered to the application programs in order of their final sequence numbers.

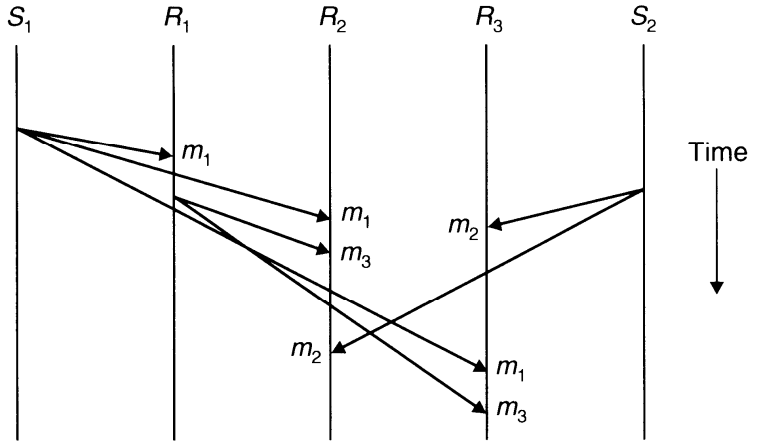
- **Causal Ordering**
  - For some applications consistent-ordering semantics is not necessary and even weaker semantics is acceptable. Therefore, an application can have better performance if the message-passing system used supports a weaker ordering semantics that is acceptable to the application. One such weaker ordering semantics that is acceptable to many applications is the causal ordering semantics. This semantics ensures that if the event of sending one message is causally related to the event of sending another message, the two messages are delivered to all receivers in the correct order. However, if two message-sending events are not causally related, the two messages may be delivered to the receivers in any order. Two message-sending events are said to be causally related if they are correlated by the happened-before relation.
  - One method for implementing causal-ordering semantics are the CBCAST protocol of the ISIS system. It assumes broadcasting of all messages to group members and works as follows:

1. Each member process of a group maintains a vector of n components, where n is the total number of members in the group. Each member is assigned a sequence number from 0 to n-1, and the i^{th} component of the vectors corresponds to the member with sequence number i. In particular, the value of i^{th} component of a member's vector is equal to the number of the last message received in sequence by this member from member i.
2. To send a message, a process increments the value of its own component in its own vector and sends the vector as part of the message.
3. When the message arrives at a receiver process's site, it is buffered by the runtime system. The runtime system tests the two conditions given below to decide whether the message can be delivered to the user process or its delivery must be delayed to ensure causal-ordering semantics. Let S be the vector of the sender process that is attached to the message and R be the vector of the receiver process. Also, let i be the sequence number of the sender process. Then the two conditions to be tested are:

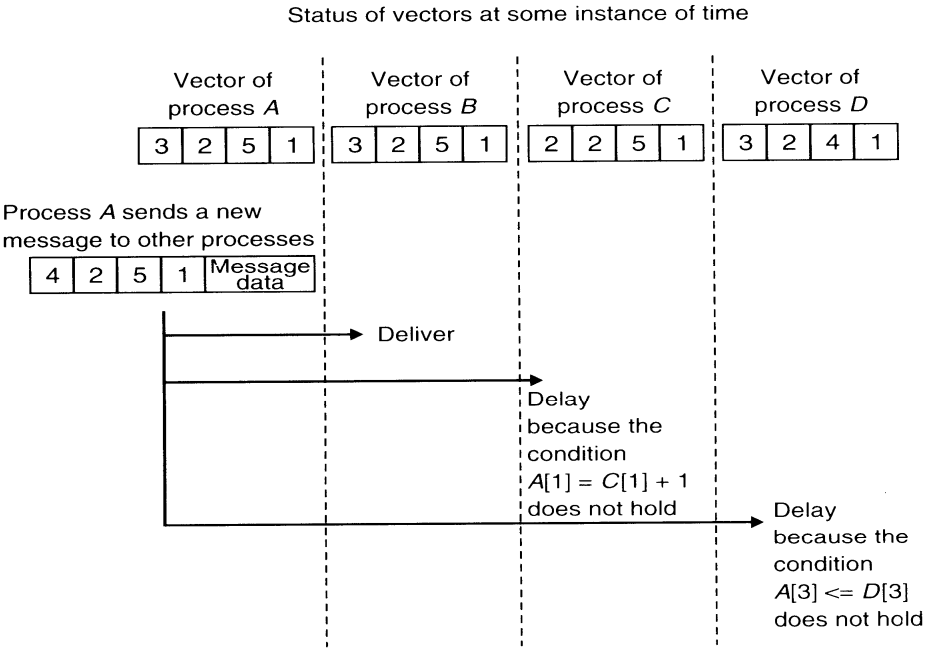
a. S [i] = R [i] +1

b. S [j] <= R [j] for all j not equal i

- The first condition ensures that the receiver has not missed any message from the sender. This test is needed because two messages from the same sender are always causally related. The second condition ensures that the sender has not received any messages that the receiver has not yet received. This test is needed to make sure that the sender's message is not causally related to a message missed by the receiver.
- If the message passes these two tests, the runtime system delivers it to the user process. Otherwise, the message is left in the buffer and the test is carried out again for it when a new message arrives.



Causal ordering of messages



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

SCSX 1028 DISTRIBUTED COMPUTING

UNIT II – REMOTE PROCEDURE CALLS

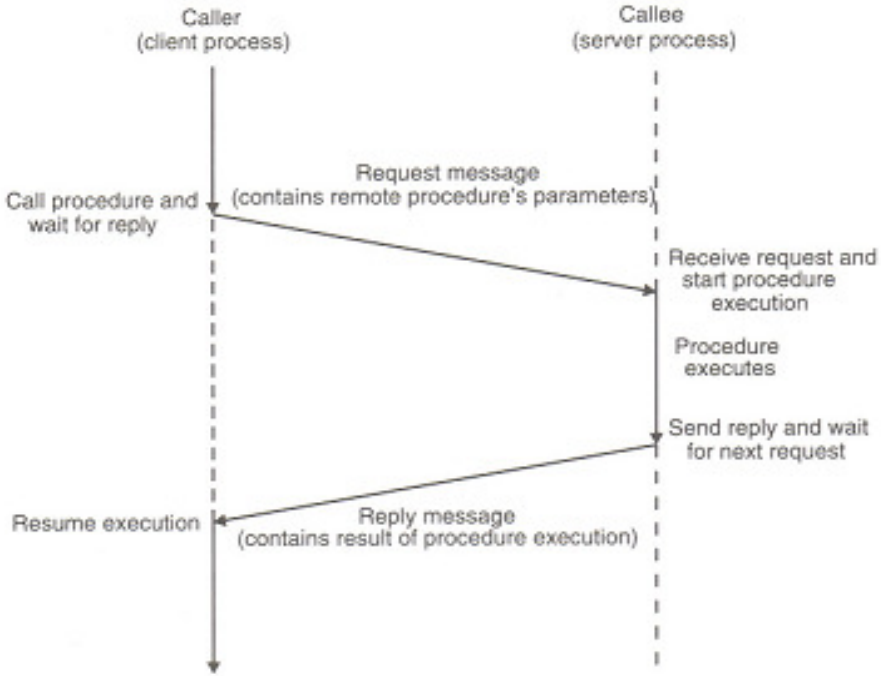
INTRODUCTION TO RPC

- A Remote Procedure Call (RPC) is an inter-process communication that allows a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.
- It further aims at hiding most of the intricacies of message passing and is idle for client-server application.
- RPC allows programs to call procedures located on other machines. But the procedures 'send' and 'receive' do not conceal the communication which leads to achieving access transparence in distributed systems.
- Example: when process A calls a procedure on B, the calling process on A is suspended and the execution of the called procedure takes place. (PS: function, method, procedure differs, stub, 5 state process model definition)
- Information can be transported in the form of parameters and can come back with procedure result. No message passing is visible to the programmer. As calling and called procedures exist on different machines, they execute in different address spaces, the parameters and result should be identical and if machines crash during communication, it causes problems.

THE RPC MODEL

- The RPC model is similar to the well known and well understood procedure call model used for the transfer of control and data within a program in the following manner:
    - For making a procedure call, the caller places arguments to the procedure in some well specified location
    - Control is then transferred to the sequence of instructions that constitutes the body of the procedure.
    - The procedure body is executed in a newly created execution environment that includes copies of the arguments given in the calling instruction.
    - After the procedure's execution is over, control returns to the calling point, possibly returning a result.

- The RPC mechanism is an extension of the procedure call mechanism in the sense that it enables a call to be made to a procedure that does not reside in the address space of the calling process.
- The called procedure (Remote Procedure) may be on the same computer or on a different computer.
- Since the caller and the callee processes have disjoint address spaces, the remote procedure has no access to data and variables of the caller's environment.
- Therefore the RPC facility uses a message passing scheme for information exchange between the caller and the callee processes.



- When a remote procedure call is made, the caller and the callee processes interact in the following manner:
    - The caller (Client Process) sends a call (request) message to the callee (server process) and waits (blocks) for a reply message. The request message contains the remote procedure's parameters.
    - The server process executes the procedure and then returns the result of procedure execution in a reply message to the client process.
    - Once the reply message is received, the result of procedure execution is extracted and the caller's execution is resumed.
- In this RPC model, only one of the two processes is active at any given time. However, in general, the RPC protocol makes no restrictions on the concurrency model implemented.

TRANSPARENCY OF RPC

- A major issue in the design of an RPC facility is its transparency property. A transparent RPC mechanism is one in which local procedures and remote procedures are (effective) indistinguishable to programmers. This requires the following two types of transparencies:
  1. **Syntactic transparency** means that a remote procedure call should have exactly the same syntax as a local procedure call.
  2. **Semantic transparency** means that the semantics of a remote procedure call are identical to those of a local procedure call.
- It is not very difficult to achieve syntactic transparency of an RPC mechanism, and we have seen that the semantics of remote procedure calls are also analogous to that of local procedure calls for most parts:
  - The calling process is suspended until the called procedure returns.
  - The caller can pass arguments to the called procedure (remote procedure).
  - The called procedure (remote procedure) can return results to the caller.
- Unfortunately, achieving exactly the same semantics for remote procedure calls as for local procedure calls is close to impossible. This is mainly because of the following differences between remote procedure calls and local procedure calls.
  1. Unlike local procedure calls, with remote procedure calls, the called procedure is executed in an address space that is disjoint from the calling program's address space. Due to this reason, the called (remote) procedure cannot have access to any variables or data values in the calling program's environment.
  2. Remote procedure calls are more vulnerable to failure than local procedure calls, since they involve two different processes and possibly a network and two different computers. Therefore, programs that make use of remote procedure calls must have the capability of handling even those errors that cannot occur in local procedure calls.
  3. Remote procedure calls consume much more time (100 – 1000 times more) than local procedure calls. This is mainly due to the involvement of a communication network in RPCs. Therefore, applications using RPCs must also have the capability to handle the long delays that may possibly occur due to network congestion.

IMPLEMENTING RPC MECHANISM

- To achieve the goal of semantic transparency, the implementation of an RPC mechanism is based on the concept of stubs, which provide a perfectly normal (local) procedure call abstraction by concealing from programs the interface to the underlying RPC system.
- We saw that an RPC involves a client process and a server process. Therefore, to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes.
- Moreover, to hide the existence and functional details of the underlying network, an RPC communication package (known as RPCRuntime) is used on both the client and server sides.
- Thus, implementation of an RPC mechanism usually involves the following five elements of program [Birrell and Nelson 1984].
    - The client
    - The client stub
    - The RPCRuntime
    - The server stub
    - The server
- The interaction between them is shown in Figure.
- The client, the client stub, and one instance of RPCRuntime execute on the client machine, while the server, the server stub, and another instance of RPCRuntime execute on the server machine. The job of each of these elements is described below.

Client:

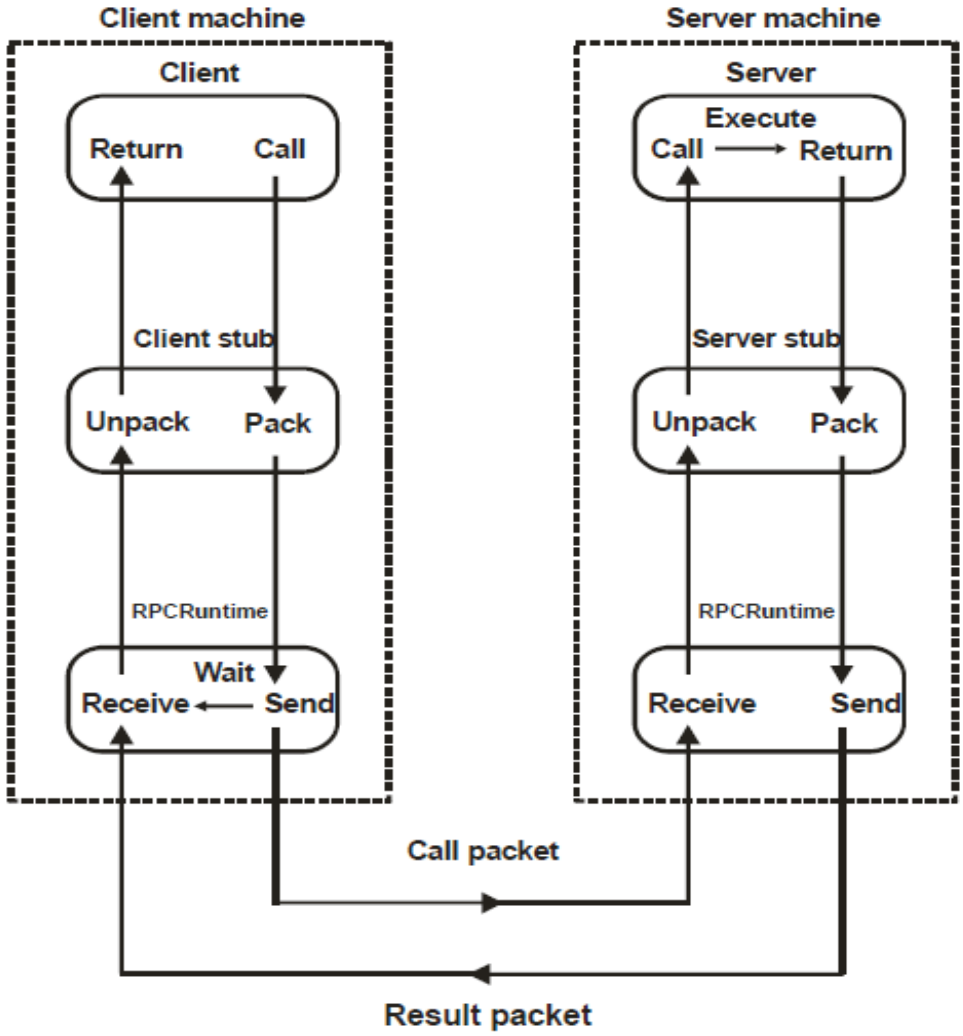
- The client is a user process that intitiates a remote procedure call.
- To make a remote procedure call, the client makes a perfectly normal local call that invokes a corresponding procedure in the client stub.

Client Stub:

- The client stub is responsible for carrying out the following two tasks :
    - On receipt of a call request from the client, it packs a specification of the target procedure and the arguments into a message and then asks the local RPCRuntime to send it to the server stub.
    - On receipt of the result of procedure execution, it unpacks the result and passes it to the client.

RPCRuntime:

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



Server Stub:

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

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

Server:

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

STUB GENERATION

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

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

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

- Similarly, an interface definition also has information about type definitions, enumerated types, and defined constants that each side uses to manipulate data from RPC calls making it unnecessary for both the client and the server to store this information separately.
- A server program that implements procedures in an interface is said to export the interface and a client program that calls procedures from an interface is said to import the interface. When writing a distributed application, a programmer first writes an interface definition using the IDL.
- He or she can then write the client program that imports the interface and the server program that exports the interface.
- The interface definition is processed using an IDL computer to generate components that can be combined with client and server programs, without making any changes to the existing compliers.

RPC MESSAGES

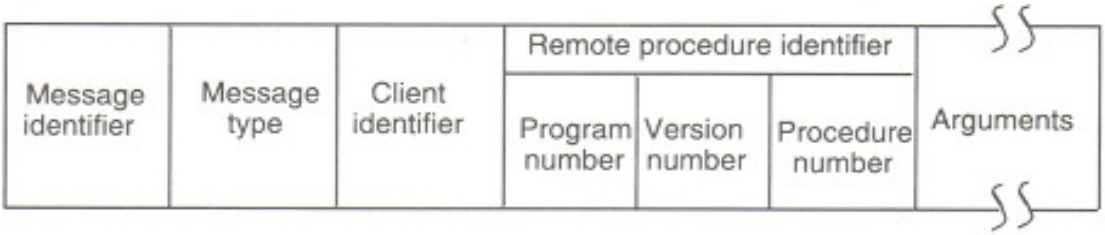
- Any remote procedure call involves a client process and a server process that are possibly located on different computers. The mode of interaction between the client and server is that the client asks the server to execute a remote procedure and the server returns the result of execution of the concerned procedure to the client.
- Based on this mode of interaction, the two types of messages involved in the implementation of an RPC system are as follows :
    1. **Call messages** that are sent by the client to the server for requesting execution of a particular remote procedure.
    2. **Reply messages** that are sent by the server to the client for returning the result of remote procedure execution.
- The protocol of the concerned RPC system defines the format of these two types of message. Normally, an RPC protocol is independent of transport protocols.
- That is, RPC does not care how a message is passed from one process to another. Therefore an RPC protocol deals only with the specification and interpretation of these two types of messages.

Call Messages:

- Since a call message is used to request execution of a particular remote procedure the two basic components necessary in a call message are as follows :
    1. The identification information of the remote procedure to be executed.

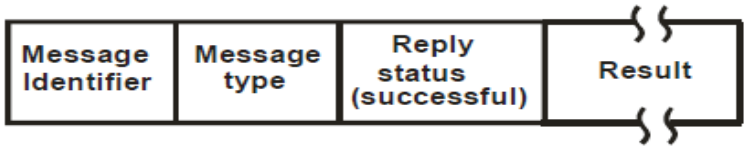
2. The arguments necessary for the execution of the procedure. In addition to these two fields, a call message normally has the following fields.
3. A message identification field that consists of a sequence number. This field is useful of two ways – for identifying lost messages and duplicate messages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

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

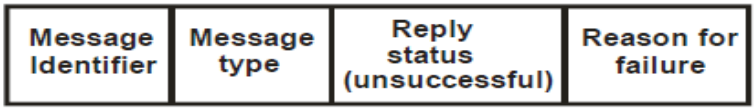


Reply Messages:

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



(a)



(b)

A typical RPC reply message format : (a) a successful reply message format; (b) an unsuccessful reply message format

MARSHALING ARGUMENTS AND RESULTS

- Implementation of remote procedure calls involves the transfer of arguments from the client process to the server process and the transfer of results from the server process to the client process.
- These arguments and results are basically language-level data structures (program objects), which are transferred in the form of message data between the two computers involved in the call.
- The transfer of message data between two computers requires encoding and decoding of the message data. For RPC this operation is known as marshaling and basically involves the following actions.
    1. Taking the arguments (of a client process) or the result (of a server process) that will form the message data to be set to the remote process.
    2. Encoding the message data of step 1 above on the sender's computer. This encoding process involves the conversion of program objects into a stream form that is suitable for transmission and placing them into a message buffer.
    3. Decoding of the message data on the receiver's computer. This decoding process involves the reconstruction of program objects from the message data that was received in stream form.
- In order that encoding and decoding of an RPC message can be performed successfully, the order and the representation method (tagged or untagged) used to marshal arguments and results must be known to both the client and the server of the RPC.
- This provides a degree of type safety between a client a server because the server will not accept a call from a client until the client uses the same interface definition as the server. Type safety is of particular importance to servers since it allows them to survive against corrupt call requests.
- The marshaling process must reflect the structure of all types of program objects used in the concerned language. These include primitive types, structured types, and user defined types.
- Marshaling procedures may be classified into two groups :
    1. Those provided as a part of the RPC software. Normally marshaling procedures for scalar data types, together with procedures to marshal compound types built from the scalar ones, fall in this group.

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

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

SERVER MANAGEMENT

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

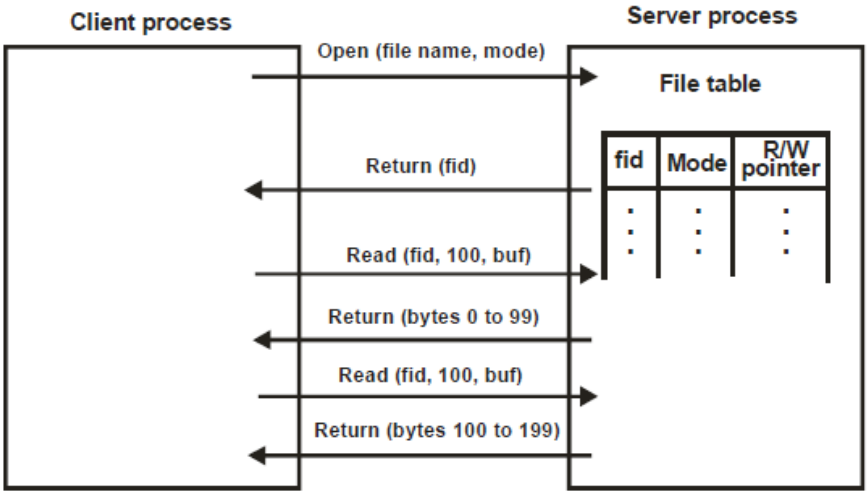
Server Implementation:

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

Stateful Servers:

- A stateful server maintains clients' state information from one remote procedure call to the next. That is, in case of two subsequent calls by a client to a stateful server, some state information pertaining to the service performed for the client as a result of the first call execution is stored by the server process.
- These clients' state information is subsequently used at the time of executing the second call.
- For example, let us consider a server for byte-stream files that allows the following operations on files :
- **Open (filename, mode):** This operation is used to open a file identified by filename in the specified mode. When the server executes this operation, it creates an entry for this file in a file-table that it uses for maintaining the file state information of all the open files. The file state information normally consists of the identifier of the file, the open mode, and the current position of a nonnegative integer pointer, called the read write pointer. When a file is opened, its read-write pointer is set to zero and the server returns to the client a file identifier (fid), which is used by the client for subsequent accesses to that file.
- **Read (fid, n, buffer):** This operation is used to get n bytes of data from the file identified by fid into the buffer named buffer. When the server executes this operation, it returns to the client n bytes of file data starting from the byte currently addressed by the read – write pointer and then increments the read – write pointer by n.

- **Write (fid, n, buffer):** On execution of this operation, the server takes n bytes of data from the specified buffer, writes it into the file identified by fid at the byte position currently addressed by the read – write pointer, and then increments the read – write pointer by n.
- **Seek (fid, position):** This operation causes the server to change the value of the read write pointer of the file identified by fid to the new value specified as position.
- **Close (fid):** This statement causes the server to delete from its file table the file state information of the file identified by fid.
- The file server mentioned above is stateful because it maintains the current state information for a file that has been opened for use by a client. Therefore, as shown in Fig. 3.3, after opening a file, if a client makes two subsequent Read (fig, 100, buf), calls, the first call will return the first 100 bytes (bytes 0 – 99) and the second call will return the next 100 bytes (bytes 100 – 199).



An example of a stateful file server

Stateless Server:

- To design an idempotent interface for reading the next record from the file, it is important that each client keeps track of its own current record position and the server is made stateless, that is, no client state should be maintained on the server side.
- Based on this idea, an idempotent procedure for reading the next record from a sequential file is ReadRecordN (Filename, N) which returns the Nth record from the specified file. In this case, the client has to correctly specify the value of n to get desired record from the file.
- However, not all non idempotent interfaces can be so easily transformed to an idempotent form.

- A stateless server does not maintain any client state information. Therefore every request from a client must be accompanied with all the necessary parameters to carry out the desired operation successfully.
- Stateless file operations are given below
- **Read (filename, position, n, buf):** On execution of this operation, the server returns n bytes of data to the client of the file identified by filename. The returned data is placed in the buffer named buf.
- **Write (filename, position, n, buf):** When the server executes the operation, it takes n bytes of data from the specified buffer and writes it into the file identified by filename. The position parameter specifies the byte position within the file from where to start writing.
- **Example**
    - **Read (filename, 0, 100,buf)**
    - **Read (filename, 100, 100, buf)**
- **Refer Book Page No. 180: Example diagram for stateless file server**

Server Creation Semantics:

- A remote procedure call made by a client process lies in a server process that is totally independent of the client process.
- Independent means that the client and server processes have separate life time.
- Based on the time duration for which RPC servers survive, they may be classified as
    - Instance-per-call Servers
    - Instance-per-session/transaction Servers
    - Persistent Servers

Instance-per-call Servers:

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

Instance-per-session/transaction Servers:

- Servers belonging to this category exist for the entire session for which a client and a server interact.
- Since a server of this type exists for the entire session, it can maintain intercall state information.

- Overhead involved in server creation and destruction is also minimized.
- Normally there is a server manager for each type of service.

Persistent Servers:

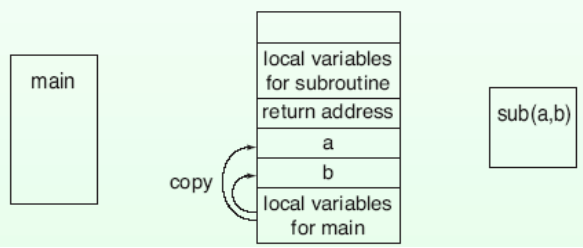
- A persistent server generally remains in existence indefinitely.
- A persistent server is usually shared by many clients.
- Servers of this type are usually created and installed before the clients use them.
- Persistent servers may also be used to improve the overall performance and reliability of the system.

PARAMETER PASSING SEMANTICS

- When a procedure is called, parameters are passed to the procedure as the arguments. There are three methods to pass the parameters.
    - call-by-value
    - call-by-reference
    - call-by-copy/restore

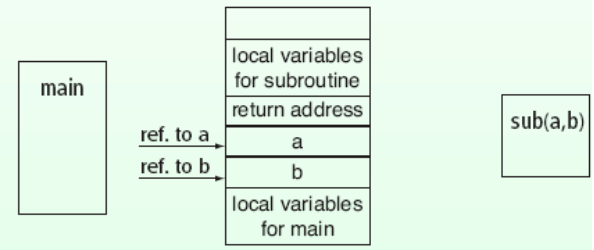
Call By Value:

- The values of the arguments are copied to the stack and passed to the procedure.
- The called procedure may modify these, but the modifications do not affect



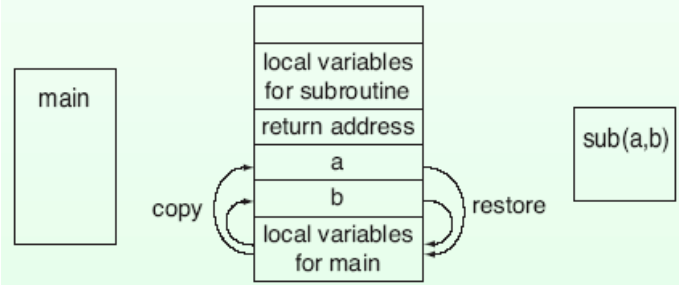
Call By Reference:

- The memory addresses of the variables corresponding to the arguments are put into the stack and passed to the procedure.
- Since these are memory addresses, the original values at the calling side are changed if modified by the called procedure.



Call By Copy / Restore:

- The values of the arguments are copied to the stack and passed to the procedure.
- When the processing of the procedure completes, the values are copied back to the original values at the calling side.
- If parameter values are changed in the subprogram, the values in the calling program are also affected.



CALL SEMANTICS

- In RPC, the caller and the callee processea are possibly located on different nodes. Thus it is possible to have failure either for the caller or for the callee.
- Therefore the normal functioning of an RPC may get distrupted due to one or more of the following reasons:
    - The call message gets lost.
    - The response message gets lost.
    - The callee node crashes and is restarted.
    - The caller node crashes and is restarted.
- Failure handling code is generally a part of RPCRuntime.
- The call semantics of an RPC system determines, how ofter the remote procedure may be executed under fault conditions.
- The following are the different types of call semantics used in RPC system.
    - **Possibly or May be call semantics:** This is the weakest semantics, not appropriate to RPC. In this method, to prevent the caller from waiting indefinitely for a response from the callee, a timeout mechanism is used. That is, the caller waits until a predetermined timeout period and then continues with its execution.
    - **Last one call semantics:** This method uses the idea of retransmitting the call message based on timeouts until a response is received by the caller. That is, the calling of the remote procedure by the caller, the execution of the procedure by the callee, and the

return of the result to the caller will be repeated until the result of the procedure execution is received by the caller.

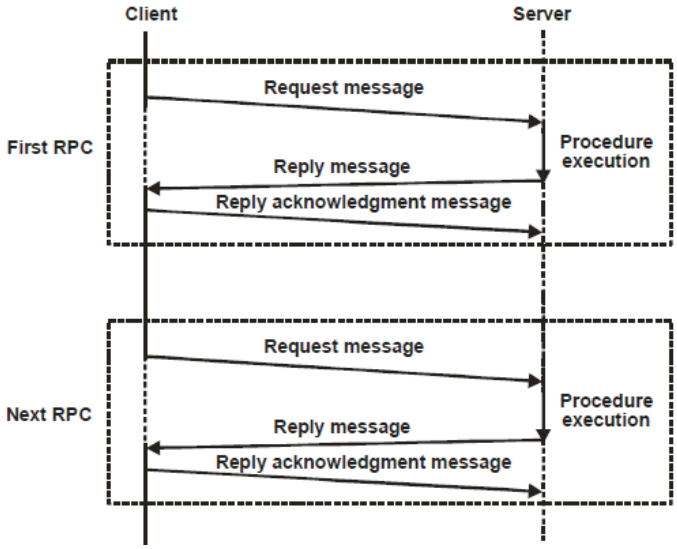
- **Last of many call semantics:** This is similar to the last one call semantics except that the orphan calls are neglected. A simple way to neglect orphan calls is to used call identifiers to uniquely identify each call. When a call is repeated, it is assigned a new call identifier. Each response message has the corresponding call identifier. A caller accepts the response only if the call identifier is associated with the identifier of the most recently repeated call, otherwise it ignores the response message.
- **Atleast once call semantics:** It just guarantees that the call is executed one or more times but does not specify which results are returned to the caller. It can be implemented simply by using timeout based retransmission.
- **Exactly once call semantics:** This is the strongest and the most desireable call semantics because it eliminated the possibility of a procedure being executed more than once, no matter how many times a call is retransmitted.

COMMUNICATION PROTOCOLS FOR RPCS

- Different systems, developed on the basis of remote procedure calls, have different IPC requirements.
- Based on the needs of different systems, several communication protocols have been proposed for use in RPCs. A brief description of these protocols is given below.
- **The Request Protocol:**
  - This protocol is also known as the R (request) protocol. It is used in RPCs in which the called procedure has nothing to return as the result of procedure execution and the client requires no confirmation that the procedure has been executed.
  - Since no acknowledgement or reply message is involved in this protocol, only one message per call is transmitted.
  - The client proceeds immediately after sending the request message as there is no need to wait for the reply message.
  - **Diagram Refer Book Page No. 188.**
  - An RPC that used the R Protocol is called asynchronous RPC.
- **The Requst / Reply Protocol:**
  - This protocol is also known as RR (Request / Reply) protocol. It is useful for the design of systems involving simple RPCs.

- A simple RPC is one in which all the arguments as well as all the results fit in a single packet buffer and the duration of a call and the interval between calls are both short.
- This protocol is based on the idea of using implicit acknowledgement to eliminate explicit acknowledgement messages.
- It does not possess failure handling capabilities.
- **Diagram Refer Book Page No. 190.**

- **The Request / Reply / Acknowledge-Reply Protocol:**



The request / reply / acknowledge reply (RRA) protocol

- This protocol is also known as the RRA (Request / Reply / Acknowledgement-reply) protocol.
- RRA protocol involves the transmission of three messages per call.
- In the RRA protocol, there is a possibility that the acknowledgement message may itself get lost.
- Therefore implementation of the RRA protocol requires that the unique message identifiers associated with request messages must be ordered.
- Each reply message contains the message identifier of the corresponding request message, and each acknowledgement message also contains the same message identifier.
- This helps in matching a reply with its corresponding request and an acknowledgement with its corresponding reply.
- A client acknowledges a reply message only if it has received the replies to all the requests previous to the request corresponding to this reply.

- Thus an acknowledgement message is interpreted as acknowledging the receipt of all reply messages corresponding to the request messages with lower message identifiers. Therefore, the loss of an acknowledgement message is harmless.

COMPLICATED RPCs

- The following are the two types of RPCs as complicated :
  1. RPCs involving long-duration calls or large gaps between calls.
  2. RPCs involving arguments and / or results that are too large to fit in a signle datagram packet.
- Different protocols are used for handling these two types of complicated RPCs.

CLIENT – SERVER BINDING

- It is necessary for a client to know the location of the server, before a remote procedure call can take place between them.
- The process by which a client becomes associated with a server, so that calls can take place is known as **binding**.
- The client server binding process involves the following issues
  - How does a client specify a server to which it wants to get bound?
  - How does the binding process locate the specified server?
  - When is it proper to bind a client to a server?
  - Is it possible for a client to change binding during execution?
  - Can a client be simultaneously bound to multiple servers that provide the same service?

Server Naming:

- The specification by a client of a server with which it wants to communicate is primarily a naming issue.
- RPC uses interface names for this purpose.
- An interface name has two parts – a type and an instance.
- Type specifies the interface itself and instance specifies a server providing the services within that interface.
- Type part of an interface usually has a version number field to distinguish between old and new versions of the interface.
- The interface name semantics are based on an arrangement between the exporter and the importer. Therefore interface names are created by the users, they are not dictated by the RPC packages.

Server Locating:

- The interface name of a server is its unique identifier.
- Thus when a client specifies the interface name of a server for making remote procedure call, the server must be located before the client's request message can be send to it.
- The two most commonly used methods are as follows.
    - **Broadcasting:** In this method, a message to locate the desired server is broadcast to all the nodes from the client node. The nodes on which the desired server is located return a response message. The desired server may be replicated on several nodes, so the client node will receive a response from all these nodes. Normally the first response received at the client node is given to the client process and all subsequent responses are discarded.
    - **Binding Agent:** A binding agent is basically a name server used to bind a client to a server by providing the client with the location information of the desired server. In this method, binding agent maintains a binding table, which is a mapping of a server's interface name to its location. **Refer Book Page No. 195 for diagram of binding mechanism for locating a server.**
    - A binding agent interface usually has three primitives:
        - **Register** is used by a server to register itself with the binding agent.
        - **Deregister** is used by a server to deregister itself with the binding agent.
        - **Lookup** is used by a client to locate a server.

Binding Time:

- A client may be bound to a server at compile time, at link time or at call time.
- **Binding at Compile Time:**
    - In this method, the client and server modules are programmed as if they were intended to be linked together.
    - For example, the server's network address can be compiled into the client code by the programmer and then it can be found by looking up the server's name in a file.
- **Binding at Link Time:**
    - In this method, a server process exports its service by registering itself with the binding agent as part of its initialization process.
    - A client then makes an import request to the binding agent for the service before making a call.

- The binding agent binds the client and the server by returning to the client the server's handle (details that are necessary to make a call to the server)

- **Binding at Call Time:**
  - In this method, a client is bound to a server at the time when it calls the server for the first time during its execution.
  - A commonly used approach for binding at call time is the indirect call method. **Refer Book Page No. 197 for diagram of binding at call time.**

Changing Bindings:

- The flexibility provided by a system to change bindings dynamically is very useful from a reliability point of view.
- Binding is a connection establishment between a client and a server.
- The client or server of connection may wish to change the binding at some instance of time dute to some change in the system state.
- For example, a client willing to get a request serviced by any one of the multiple servers for that service may be programmed to change a binding to another server of the same type when a call to the already connected server fails.

Multiple simultaneous Bindings:

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

SECURITY

- Some implementations of RPC include facilities for client and server authentication as well as for providing encryption – based security for calls.
- For example, callers are given a guarantee of the identity of the callee, and vice versa, by using the authentication service of Grapevine.

- For full end-to-end encryption of calls and results, the federal data encryption standard is used in. The encryption techniques provide protection from eavesdropping (and conceal pattern of data) and detect attempts at modification, replay, or creation of calls.
- In other implementations of RPC that do not include security facilities, the arguments and results of RPC are readable by anyone monitoring communications between the caller and the callee.
- Therefore, in this case, if security is desired, the user must implement his or her own authentication and data encryption mechanisms.
- When designing an application, the user should consider the following security issues related with the communication of messages :
    - Is the authentication of the server by the client required?
    - Is the authentication of the client by the server required when the result is returned?
    - Is it all right if the arguments and results of the RPC are accessible to users other than the caller and the callee?

SOME SPECIAL TYPES OF RPCs

Callback RPC:

- In the usual RPC protocol, the caller and callee processes have a client – server relationship. Unlike this, the callback RPC facilitates a peer-to-peer paradigm among the participating processes. It allows a process to be both a client and a server.
- Callback RPC facility is very useful in certain distributed applications.
- For example, remotely processed interactive applications that need user input from time to time or under special conditions for further processing require this type of facility.
- In such applications, the client process makes an RPC to the concerned server process, and during procedure execution for the client, the server process makes a callback RPC to the client process.
- The client process takes necessary action based on the server's request and returns a reply for the call back RPC to the server process.
- On receiving this reply, the server resumes the execution of the procedure and finally returns the result of the initial call to the client.
- Note that the server may make several callbacks to the client before returning the result of the initial call to the client process.

- The ability for a server to call its client back is very important, and care is needed in the design of RPC protocols to ensure that it is possible. In particular, to provide callback RPC facility, the following are necessary :
    - Providing the server with the client's handle
    - Making the client process wait for the callback RPC
    - Handling callback deadlocks
- **Refer Book Page No. 200 for diagram of the call back RPC**

Broadcast RPC:

- In broadcast RPC, a client's request is broadcast on the network and is processed by all the servers that have the concerned procedure for processing that request. The client waits for and receives numerous replies.
- A broadcase RPC mechanism may use on of the following two methods.
    - The client has to use a special broadcast primitive to indicate that the request message has to be broadcasted. The request is sent to the binding agent, which forwards the request to all the servers registered with it.
    - The second method is to declare broadcast ports. A network port of each node is connected to a broadcast port. A network port of a node is a queuing point on that node for broadcast messages. The client of the broadcast RPC first obtains a binding for a broadcast port and then broadcasts the RPC messages by sending the message to this port.

Batch Mode RPC:

- Batch mode RPC is used to queue separate RPC requests in a transmission buffer on the client side and then send them over the network in one batch to the server.
- This helps in the following two ways
    - It reduces the overhead involved in sending each RPC request independently to the server and waiting for a response for each request.
    - Application requiring higher call rates may not be feasible with most RPC implementations. Such applications can be accommodated with the use of batch mode RPC.

RPC IN HETEROGENEOUS ENVIRONMENTS

- Heterogeneity is an important issue in the design of any distributed application.

- When designing an RPC system for a heterogeneous environment, the three common types of heterogeneity need to be considered.
- **Data Representation:** Machines having different architectures may use different data representations. Integer may be represented in 1's complement notation in one machine architecture and in 2's complement notation in another machine architecture. Floating-point representations may also vary between two different machine architectures. Therefore, an RPC system for a heterogeneous environment must be designed to take care of such differences in data representations between the architectures of client and server machines of a procedure call.
- **Transport protocol:** For better portability of applications, an RPC system must be independent of the underlying network transport protocol. This will allow distributed applications using the RPC system to be run on different networks that use different transport protocols.
- **Control protocol:** For better portability of applications, an RPC system must also be independent of the underlying network control protocol that defines control information in each transport packet to track the state of a call.
- The most commonly used approach to deal with these types of heterogeneity while designing an RPC system for a heterogeneous environment is to delay the choices of data representation, transport protocol, and control protocol until bind time.
- In conventional RPC systems, all these decisions are made when the RPC system is designed. That is, the binding mechanism of an RPC system for a heterogeneous environment is considerably richer in information than the binding mechanism used by a conventional RPC system.
- It includes mechanisms for determining which data conversion software (if any conversion is needed), which transport protocol, and which control protocol should be used between a specific client and server and returns the correct procedures to the stubs as result parameters of the binding call.
- These binding mechanism details are transparent to the users.

LIGHTWEIGHT RPC

- The Lightweight Remote Procedure Call (LRPC) was introduced by Berhsad and integrated into the Taos operating system of the DEC SRC Firefly microprocessor workstation.
- Based on the size of the kernel, operating systems may be broadly classified into two cateogies
    - Monolithic kernel operating systems

- Microkernel operating systems.

- Monolithic kernel operating systems have a large kennel, monolithic kernel is insulated from user programs by simple hardware boundaries.
- On the other hand, in microkernel operating systems, a small kernel provides only primitive operations and most of the services are provided by user-level servers.
- The servers are usually implemented as processes and can be programmed separately. Each server forms a component of the operating system and usually has its own address space.
- As compared to the monolithic kernel approach, in this approach services are provided less efficient because the various components of the operating system have to use some form of IPC to communicate with each other.
- The advantages of this approach include simplicity and flexibility. Due to modular structure, microkernel operating systems are simple and easy to design, implement, and maintain.
- In the microkernel approach, when different components of the operating system have their own address spaces, the address space of each component is said to form a domain, and messages are used for all interdomain communication.
- In this case, the communication traffic in operating systems are of two types:
  - Cross-domain, which involves communication between domains on the same machine.
  - Cross-machine, which involves communication between domains located on separate machines.
- The LRPC is a communication facility designed and optimized for cross-domain communications.
- Although conventional RPC systems can be used for both cross-domain and cross machine communications, it is observed that the use of conventional RPC systems of crossdomain communications, which dominate cross-machine communications, incurs an unnecessarily high cost.
- Based on these observations, Bershad et al. Designed the LRPC facility for crossdomain communications, which has better performance than conventional RPC systems.
- LPRC is safe and transparent and represents a viable communication alternative for microkernel operating systems.
- To achieve better performance than conventional RPC systems, the four techniques described below are used by LRPC.

Simple Control Transfer :

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

Simple Data Transfer:

- As compared to traditional RPC systems, LRPC reduces the cost of data transfer by performing fewer copies of the data during its transfer from one domain to another.
- **Refer Book Page No. 206 for diagram of data transfer.**
- In traditional cross domain RPC, an argument normally has to be copied four times
    - From the client's stack to the RPC message.
    - From the message in the client domain to the message in the kernel domain.
    - From the message in the kernel domain to the message in the server domain.
    - From the message in the server domain to the server's stack.
- To simplify this data transfer operation, LRPC uses a shared argument stack that is accessible both by the client and the server. Therefore the same argument in LRPC can be copied only once – from the client's stack to the shared argument stack. The server uses the argument from the argument stack.

Simple Stubs:

- Cross domain and cross machine calls are usually made transparent to the stubs.

- The use of simple models of control and data transfer in the LRPC facilitates the generation of highly optimized stubs.
- A three layered communication protocol is defined for each procedure in and LRPC interface
    - End to end, described by the calling conventions of the programming language and architectures.
    - Stub to stub, implemented by the stubs themselves.
    - Domain to domain, implemented by the kernel.
- With this arrangement, a simple LRPC needs only one formal procedure call and two returns.

Design for Concurrency:

- When the node of the client and server processes of an LRPC has multiple processors with a shared memory, special mechanisms are used to achieve higher call throughput and lower call latency.
- LRPC achieves a factor-of-three performance improvement over more traditional approaches.
- Thus LRPC reduces the cost of cross domain communication to nearly the lower bound imposed by conventional hardware.

SCSX 1028 DISTRIBUTED COMPUTING

UNIT III

DISTRIBUTED SHARED MEMORY

INTRODUCTION

Two basic paradigms for inter process communication are

- Shared memory paradigm
- Message passing paradigm

Message passing paradigm has two basic primitives for interprocess communication

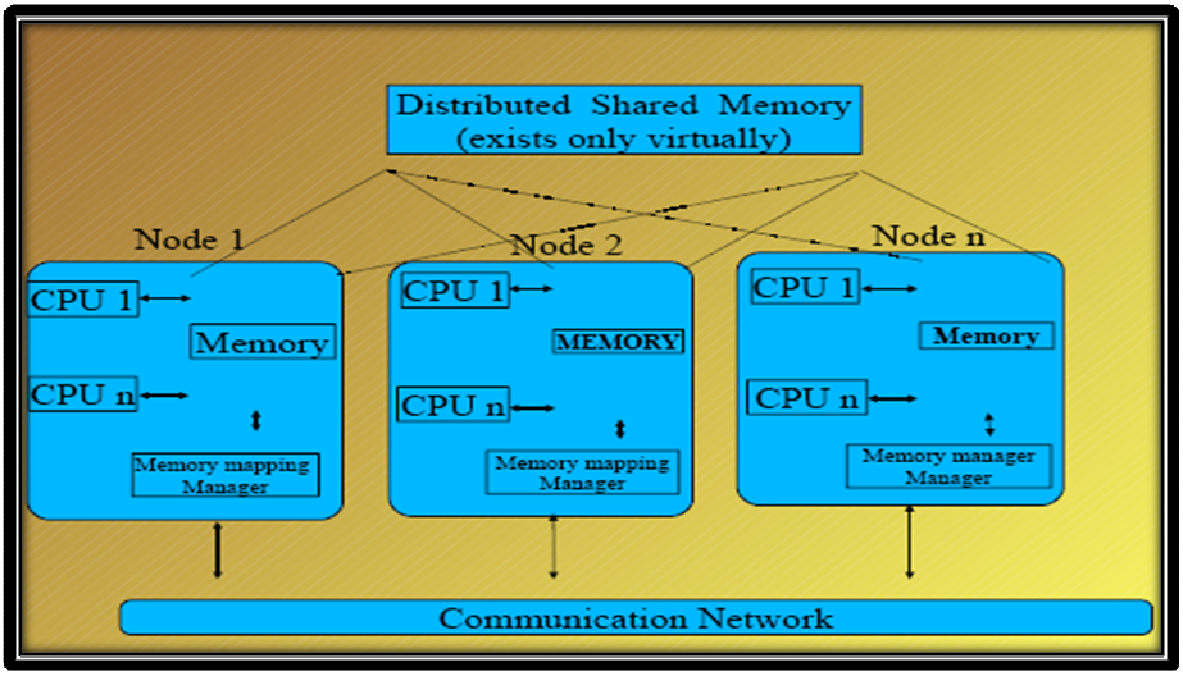
- Send(recipient, data)
- Receive(data)

Shared memory paradigm has two basic primitives for interprocess communication

- Data=Read(address)
- Write(address, data)

Shared memory exists only virtually. Similar concept to virtual memory.DSM also known as DSVM. DSM provides a virtual address space shared among processes on loosely coupled processors. DSM is basically an abstraction that integrates the local memory of different machine into a single logical entity shared by cooperating processes

ARCHITECTURE OF DSM



- Each node of the system consist of one or more CPUs and memory unit.
- Nodes are connected by high speed communication network
- Simple message passing system for nodes to exchange information
- In contrast to the shared memory in tightly coupled parallel architectures, the shared memory of DSM exist only virtually
- A software Memory mapping manager routine maps local memory to shared virtual memory
- To facilitate the Shared memory space is partitioned into blocks
- Data caching is used in DSM system to reduce network latency
- The basic unit of caching is a memory block
- When a process on a node accesses some data from a memory block of the shared memory space, the local memory mapping manager take charge for its request
- If a memory block containing the accessed data is resident in the local memory the request is satisfied by supplying the accessed data from the local memory.
- Otherwise , a network block fault is generated and the control is passed to the OS.
- The OS then sends a message to the node on which the desired memory block is located to get the block
- The missing block is migrated from the remote node to the client process's node and operating system maps into the application's address space.
- Data block keep migrating from one node to another on demand but no communication is visible to the user processes
- Copies of data cached in local memory eliminate network traffic for a memory access

DESIGN AND IMPLEMENTATION ISSUES

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

GRANULARITY

Granularity refers to the block size of DSM. The unit of sharing and the unit of data transfer across the network when a network block fault occur. Possible unit are a few word , a page or a few pages. Selecting proper block size is an important part of the design of a DSM system

STRUCTURE OF SHARED MEMORY

Structure refers to the layout of the shared data in memory.

Dependent on the type of applications that the DSM system is intended to support.

DATA LOCATION AND ACCESS

To share data in a DSM, should be possible to locate and retrieve the data accessed by a user process. DSM system must implement some form of data block locating mechanism inorder to service network data block fault

MEMORY COHERENCE AND ACCESS SYNCHRONIZATION

In a DSM system that allows replication of shared data item, copies of shared data item may simultaneously be available in the main memories of a number of nodes

To solve the memory coherence problem that deal with the consistency of a piece of shared data lying in the main memories` of two or more nodes .

Different memory coherence protocol makes different assumptions, the choice is usually dependent on the pattern of memory access.

Memory coherence protocol alone is not sufficient therefore in addition synchronization primitives, such as semaphores, event count and locks are needed to synchronize concurrent accesses to shared data

REPLACEMENT STRATEGY:

If the local memory of a node is full, a cache miss at that node implies not only a fetch of accessed data block from a remote node but also a replacement

Data block must be replaced by the new data block

THRASHING

Data block migrate between nodes on demand. Therefore if two nodes compete for write access to a single data item the corresponding data block may be transferred back and forth at such a high rate that no real work can get done.

A DSM system must use a policy to avoid this situation.

HETEROGENEITY

The DSM system built for homogeneous system need not address the heterogeneity issue

If the system is heterogeneous, must be designed to take care of heterogeneity so that it functions properly with machines having different architectures.

GRANULARITY

Most visible parameter in the design of DSM system is block size

Factors influencing block size selection:

Sending large packet of data is not much more expensive than sending small ones

Paging overhead: A process is likely to access a large region of its shared address space in a small amount of time

Therefore the paging overhead is less for large block size as compared to the paging overhead for small block size

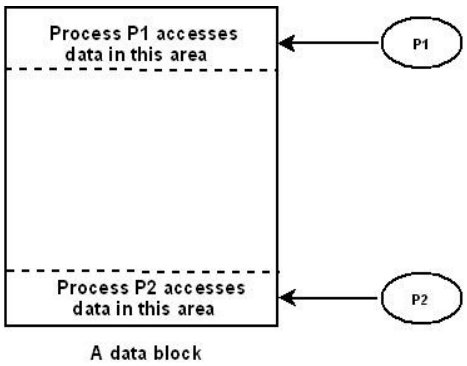
Directory size: The larger the block size, the smaller the directory

Ultimately result in reduced directory management overhead for larger block size

Thrashing: The problem of thrashing may occur when data item in the same data block are being updated by multiple node at the same time

Problem may occur with any block size, it is more likely with larger block size, is different regions in the same block may be updated by processors on different nodes, causing data block transfers that are not necessary with smaller block sizes.

False sharing:



Occur when two different processes access two unrelated variable that reside in the same data block . The larger is the block size the higher is the probability of false sharing. False sharing of a block may lead to a thrashing problem

Using page size as block size:

Relative advantage and disadvantages of small and large block size make it difficult for DSM designer to decide on a proper block size

Following advantage:

It allows the use of existing page fault schemes to trigger a DSM page fault

It allows the access right control

AS long as page can fit into a packet, Page size do not impose undue communication overhead at the time of network page fault

Page size is a suitable data entity unit with respect to memory contention

Structure of shared-memory space

Structure defines the abstract view of the shared memory space

The structure and granularity of a DSM system are closely related

Three approach:

1. No structuring
2. Structuring by data type
3. Structuring as a database

NO STRUCTURING

The shared memory space is simply a linear array of words

Advantage:

Choose any suitable page size as the unit of sharing and a fixed grain size may be used for all application

Simple and easy to design such a DSM system

STRUCTURING BY DATA TYPE

The shared memory space is structured either as a collection of variables or as collection of objects in the source language

The granularity in such DSM system is an object or a variable

Since the size of the variables and objects varies greatly, DSM system use variable grain size to match the size of the object/variable being accessed by the application.

The use of variable grain size complicates the design and implementation of this DSM system

STRUCTURING AS A DATABASE

Structure the shared memory like a database

Shared memory space is ordered as an associative memory called tuple space which is a collection of immutable tuples with typed data items in their fields

To perform update old data item in the DSM are replaced by new data item

Processes select tuples by specifying the number of their fields and their values or type

Access to shared data is nontransparent. Most system they are transparent

CONSISTENCY MODELS

Consistency requirement vary from application to application

A consistency model basically refers to the degree of consistency that has to be maintained for the shared memory data

Defined as a set of rules that application must obey if they want the DSM system to provide the degree of consistency guaranteed by the consistency model

Applications that depends on a strong consistency model may not perform correctly if executed in a system that support only a weak consistency model.

If a system support the stronger consistency model then the weaker consistency model is automatically supported but the converse is not true

TYPES OF CONSISTENCY MODELS

- Strict Consistency model
- Sequential Consistency model
- Causal consistency model
- Pipelined Random Access Memory consistency model(PRAM)
- Processor Consistency model
- Weak consistency model
- Release consistency model

STRICT CONSISTENCY MODEL

This is the strongest form of memory coherence having the most stringent consistency requirement

Value returned by a read operation on a memory address is always same as the value written by the most recent write operation to that address

All writes instantaneously become visible to all processes

Implementation of the strict consistency model requires the existence of an absolute global time

Absolute synchronization of clock of all the nodes of a distributed system is not possible

Implementation of strict consistency model for a DSM system is practically impossible

SEQUENTIAL CONSISTENCY MODEL

A shared memory system is said to support the sequential consistency model if all processes see the same order

Exact order of access operations are interleaved does not matter

If the three operations read(r1), write(w1), read(r2) are performed on a memory location in that order

Any of the orderings (r1, w1, r2), (r1, r2, w1), (w1, r1, r2), (w1, r2, r1), (r2, r1, w1), (r2, w1, r1) is acceptable provided all processes see the same ordering

The consistency requirement of the sequential consistency model is weaker than that of the strict consistency model

The sequential consistency model does not guarantee that a read operation on a particular memory address always returns the same value as return by the most recent write operation to that address.

A sequentially consistency memory provide one-copy /single-copy semantics

Sequentially consistency is acceptable by most applications

CAUSAL CONSISTENCY MODEL

All processes see only those memory reference operations in the correct order that are potentially causally related

Memory reference operations not related may be seen by different processes in different order

Memory reference operation is said to be related to another memory reference operation if one might have been influenced by the other

A shared memory is said to support the causal consistency model

Write operations that are potentially causally related must be seen in the "***same order***"

Write operations that are not potentially causally related may be seen in different orders

(w1, w2) is acceptable ; (w2,w1) is not an acceptable order

There is a necessity to keep track of the memory reference operations

This is done by Maintaining dependency graphs for memory access operations

PIPELINED RANDOM ACCESS MEMORY CONSISTENCY MODEL

Provides a weaker consistency semantics than the consistency model described so far

Ensures that all write operations performed by a single process are seen by all other processes in the order in which they were performed

All write operations performed by a single process are in a pipeline

Write operations performed by different processes can be seen by different processes in different order

If w11 and w12 are two write operations performed by a process P1 in that order, and w21 and w22 are two write operations performed by a process P2 in that order

A process P3 may see them in the order [(w11,w12), (w21,w22)] and another process P4 may see them in the order [(w21,w22), (w11,w12)]

Simple and easy to implement and also has good performance

It can be implemented by simply sequencing the write operations performed at each node

PRAM consistency all processes do not agree on the same order of memory reference operations

[(w11,w12), (w21,w22)] and [(w21,w22), (w11,w12)] are acceptable by Sequential consistency model.

[(w11,w12), (w21,w22)] PRAM accepts only this order

Processor consistency model

Very similar to PRAM model with additional restriction of memory coherence

Memory coherence means that for any memory location all processes agree on the same order of all write operations performed on the same memory location (no matter by which process they are performed) are seen by all processes in the same order

If w12 and w22 are write operations for writing the same memory location x, all processes must see them in the same order- w12 before w22 or w22 before w12

Processes P3 and P4 must see in the same order, which may be either [(w11,w12), (w21,w22)] or [(w21,w22), (w11,w12)]

WEAK CONSISTENCY MODEL

Common characteristics to many application:

1 It is not necessary to show the change in memory done by every write operation to other processes. Several write operation results can be combined and sent to other only when they need it.eg. when a process executes in a critical section the changes need not be visible until it exits the critical section

2 Isolated accesses to shared variable are rare. A process makes several access to the set of shared variables and no access at all for a long time.

Better performance can be achieved if consistency is enforced on a group of memory reference operations rather than on individual memory reference operations

The main problem is determining how the system can know that it is time to show changes to other processors.

DSM system that support the weak consistency model uses a special variable called a ***synchronization variable***

When a synchronized variable is accessed by a process the entire shared memory is synchronized by making all changes to the memory

Requirements:

All accesses to synchronization variables must obey sequential consistency semantics

All previous write operations must be completed everywhere before an access to a synchronization variable is allowed

All previous accesses to synchronization variables must be completed before access to a non synchronization variable is allowed

RELEASE CONSISTENCY MODEL

Enhancement of weak consistency model

Two operation in memory synchronization

1. All changes made by the other nodes is propagating from other nodes to process's node
2. All changes made by process propagating from the process to other nodes

The first operation is performed when process enters the critical section.

The second operation is performed when process exits the critical section

Use of two synchronization variables

1. Acquire (used to tell the system it is entering CR)
2. Release (used to tell the system it has just exited CR)

Barrier defines the end of a phase of execution of a group of concurrently executing processes

All processes in the group must complete their execution up to the barrier before any process is allowed to proceed.

It waits until all process in the group attains the barrier.

Once the last process completes the execution all the shared variables are synchronized

Barrier can be implemented by using a centralized barrier server

Before a barrier is created it must be given the count of number of processes that must be waiting on it.

Each process in the group concurrently gets executed and send the message to the server when it arrives the barrier

The barrier server sends a reply after all the sever has sent the message of reaching the barrier

Requirements:

All accesses to acquire and release synchronization variable obey processor consistency semantics

All previous acquires perform by a process must be completed successfully before the process is allowed to perform a data access operation on the memory

All previous data access operations performed by a process must be completed successfully before a release access done by the process is allowed

IMPLEMENTING SEQUENTIAL CONSISTENCY MODEL

Most commonly used model

Protocols for implementing the sequential consistency model in the DSM system depend to a great extent on whether the DSM system allows replication and/or migration of shared memory data blocks

Strategies:

- *Nonreplicated, Nonmigrating blocks (NRNMB)*
- *Nonreplicated, migrating blocks (NRMB)*
- *Replicated, migrating blocks (RMB)*
- *Replicated, Nonmigrating blocks (RNMB)*

Nonreplicated, Nonmigrating blocks (NRNMB)

Simplest strategy for implementing a sequentially consistency DSM system

Each block of the shared memory has a single copy whose location is always fixed.

All access request to a block from any node are send to the owner node of the block.

On receiving a request from a client, the memory management unit and OS s/w of the owner node perform the access request on the block and return a response to the client

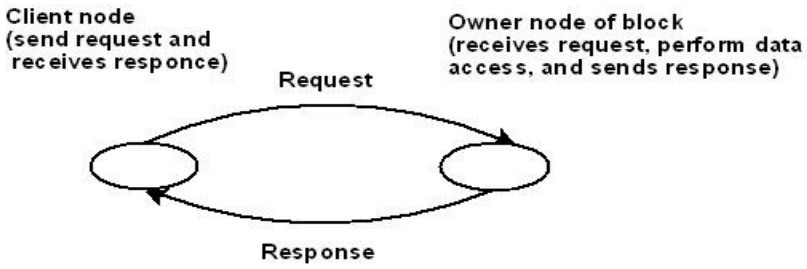


Fig 3 NRNMB strategy

Enforcing sequential consistency is simple in this case

Method is simple and easy to implement and suffers following drawback:

Serializing data access creates a bottleneck Parallelism, which is a major advantage of DSM is not possible with this method

Data locating in the NRNMB strategy:

There is a single copy of each block in the entire system

The location of a block never changes

Hence use Mapping Function to map a block to a node

Nonreplicated, migrating blocks (NRMB)

Each block of the shared memory has a single copy in the entire system

Each access to a block causes the block to migrate from its current node to the node from where it is accessed .

Owner of a block changes as soon as the block is migrate to a new node.

When a block migrate, it is removed from any local address space it has been mapped into.

In this strategy only the processes executing on one node can read or write a given data item at any one time and ensures sequential consistency

Advantage :

- No communications cost are incurred when a process accesses data currently held locally
- It allows the applications to take advantage of data access locality

Drawbacks:

- It is prone to thrashing problem

The advantage of parallelism can not be availed in this method also

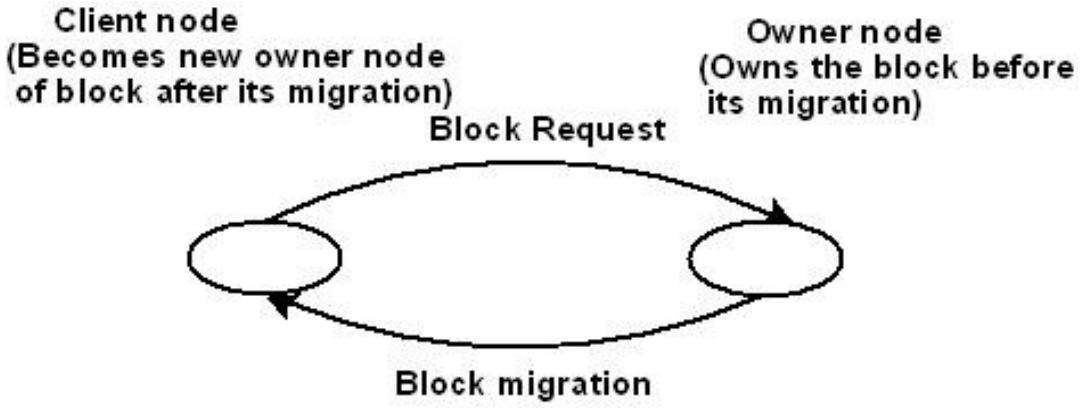


Fig 4. NRMB strategy

Data locating in the NRMB strategy

There is a single copy of each block, the location of a block keeps changing dynamically

Following method used:

1. Broadcasting
2. Centralized server algorithm
3. Fixed distributed server algorithm
4. Dynamic distributed

Broadcasting

Each node maintains an owned blocks table that contains an entry for each block for which the node is the current owner

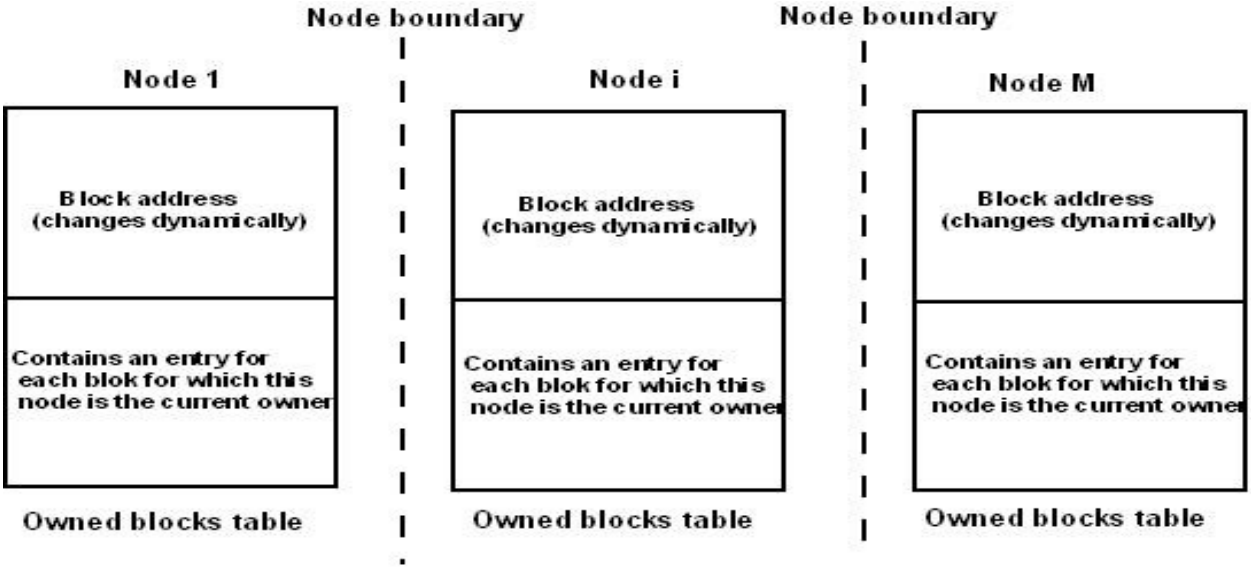


Fig 5. structure and locations of owned bloks table in the broadcasting data locating mechanism for NRMB strategy

When a fault occurs, the fault handler of the faulting node broadcasts a read/write request on the network

The node currently having the request block then responds to the broadcast request by sending the block to the requesting node.

Disadvantage:

- It does not scale well.
- When a request is broadcast, all nodes must process the request. This makes the system bottleneck

Centralized server algorithm

A centralized server maintains a block table that contains the location information for all block in the shared memory space

The location and identity of the centralized server is well known to all nodes

When fault occurs, the fault handler of the faulting node send a request for the accessed block to the centralized server.

The centralized server extract the location information of the requested block from the block table, forwards the request to that node and changes the location information in the block table.

On receiving the request , the current owner transfers the block to node N, which becomes the new owner of the block.

Drawback:

- A centralized server serializes location queries, reducing parallelism
- The failure of the centralized server will cause the DSM system to stop functioning

Fixed distributed server algorithm

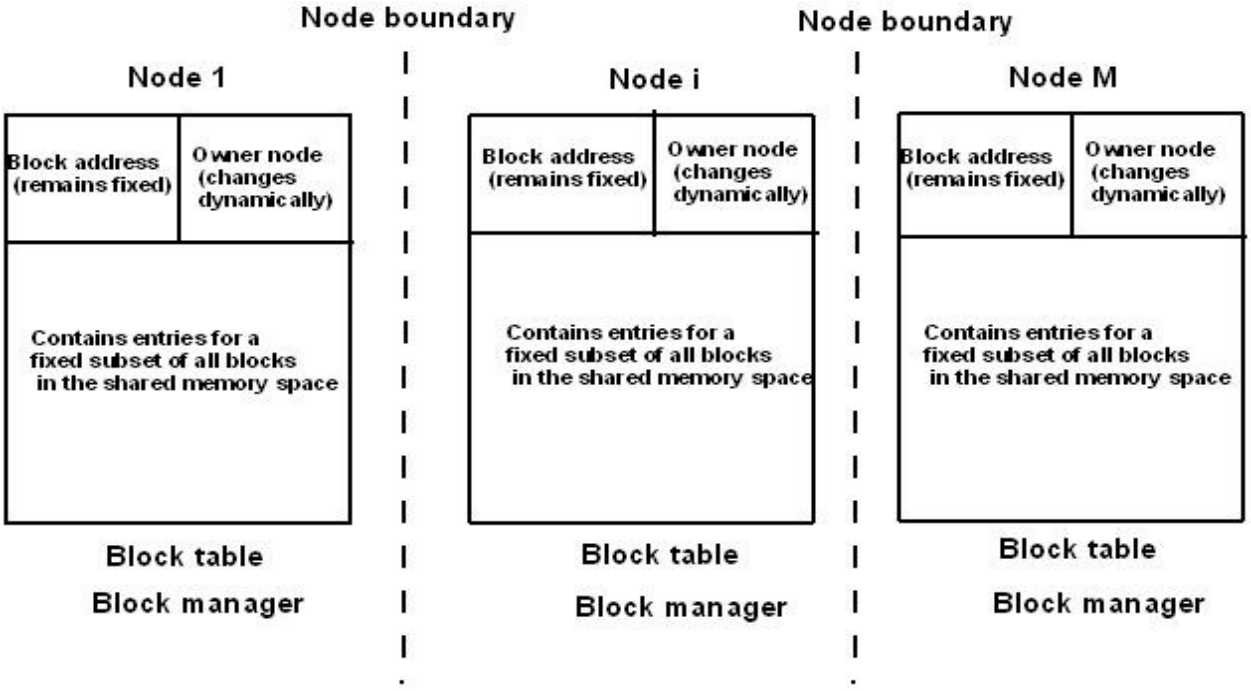


Fig 7. structure and locations of block table in the fixed distributed server data locating mechanism for NRMB strategy

Scheme is a direct extension of the centralized server scheme

It overcomes the problems of the centralized server scheme by distributing the role of the centralized server

In this , there is a block manager on several nodes, each block manager is given a pre determined data blocks to manage .

The mapping from data blocks to block manager and their corresponding node is by mapping function.

Whenever a fault occurs, the mapping functions is used by the fault handler of the faulting node to find out the node whose block manager is mapping the currently accessed block

Dynamic distributed-server algorithm

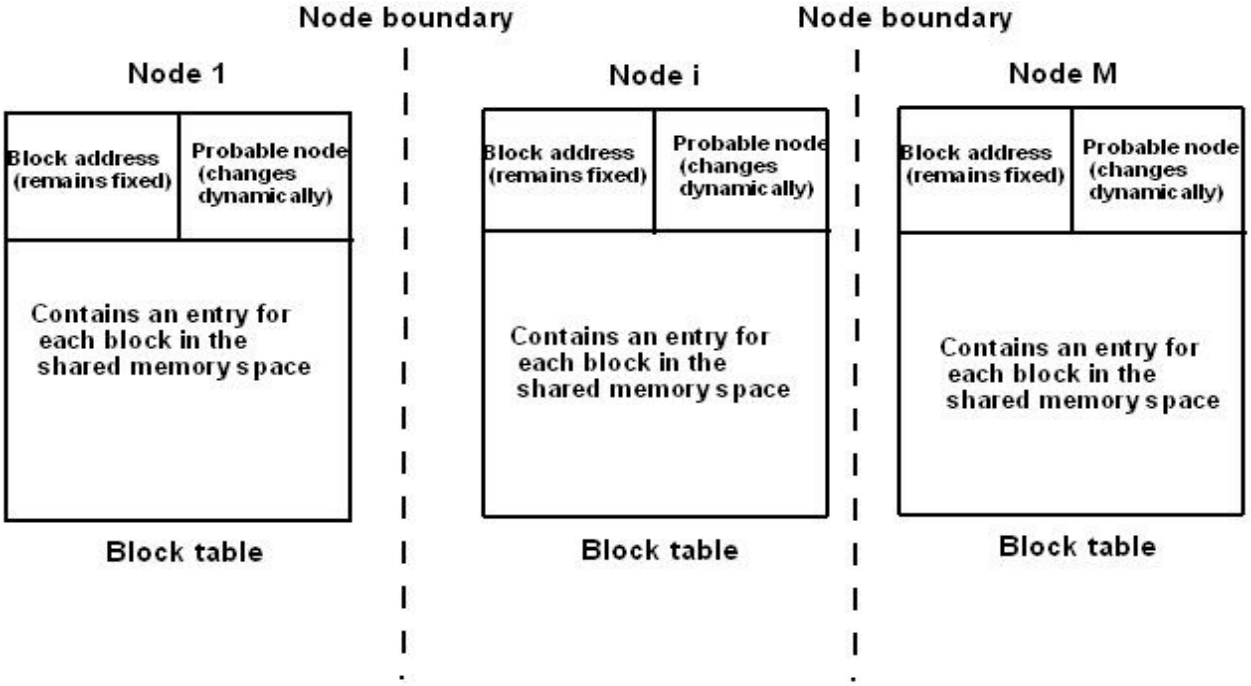


Fig 8. structure and locations of block table in the dynamic distibuted server data locating mechanism for NRMB strategy

Does not use any block manager and attempts to keep track of the ownership information of all block in each node

Each node has a block table that contains the ownership information for all block in the shared memory space.

The information in ownership field is not correct at all times, it at least provides the beginning of a sequence of node to be traversed to reach the true owner node of a block

A field gives the node a hint on the location of the owner of a block and hence is called the probable owner

When fault occurs, the faulting node extracts from its block table the node information stored in the probable owner field .

It then send a request for the block to that node.

If that node is the true owner of the block, it transfers the block to node N and updates the location information of the block in its local block table of node N.

Otherwise, it looks up its local block table, forward the request to the node indicating in the probable owner field of the entry for the block, and updates the value of this field to node N.

When node N receives the block , it become the new owner of the block.

Replicated, migrating blocks (RMB)

A major disadvantage of the non replication strategies is lack of parallelism

To increase parallelism, virtually all DSM system replicate blocks

With replicated blocks, read operation can be carried out in parallel at multiple nodes by accessing the local copy of the data.

Replication tends to increase the cost of write operation because for a write to a block all its replica must be invalidated or updated to maintain consistency

If the read/write ratio is large, the extra expense for the write operation may be more than offset by the lower average cost of the read operation

Two basic protocols that may be used for ensuring sequential consistency in this case are:

write-invalidate:

All copies of a piece of data except one are invalidated before a write can be performed on it.

When a write fault occurs fault handler copies the accessed block from one of the current node to its own node.

Invalidates all other copies of the block by sending an invalidate message containing the block address to the node having a copy of the block.

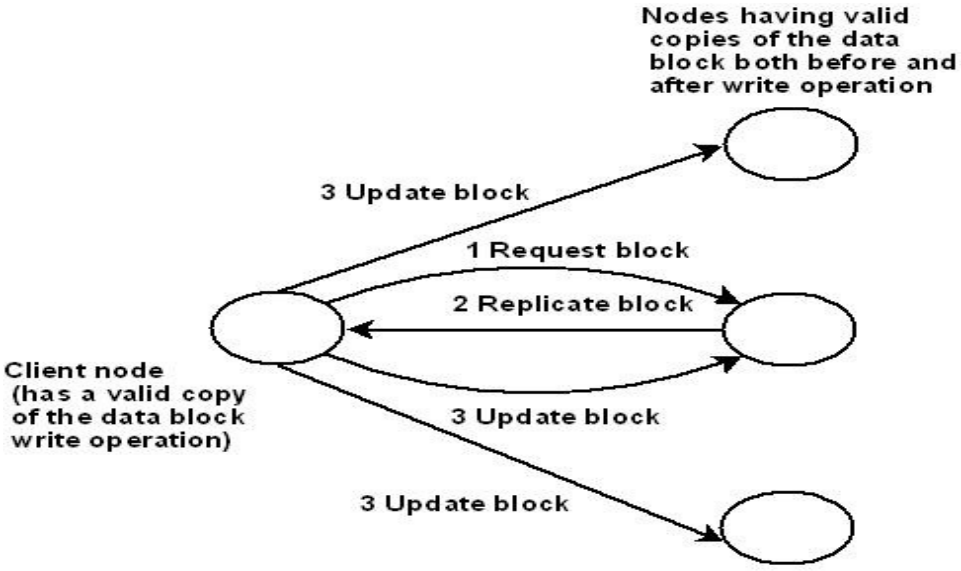


Fig 10. write update memory coherence approach for replicated, migrating blocks strategy

The node "owns" that block and can proceed with the write operation and other read/write operations until the block ownership is relinquished to some other node.

After invalidation of a block, only the node that performs the write operation on the block holds the modified version of the block

If one of the nodes that had a copy of the block before invalidation tries to perform a memory access operation (read/write) on the block after invalidation, a cache miss will occur and the fault handler of that node will have to fetch the block again from a node having a valid copy of the block, therefore the scheme achieves sequential consistency

Write-update

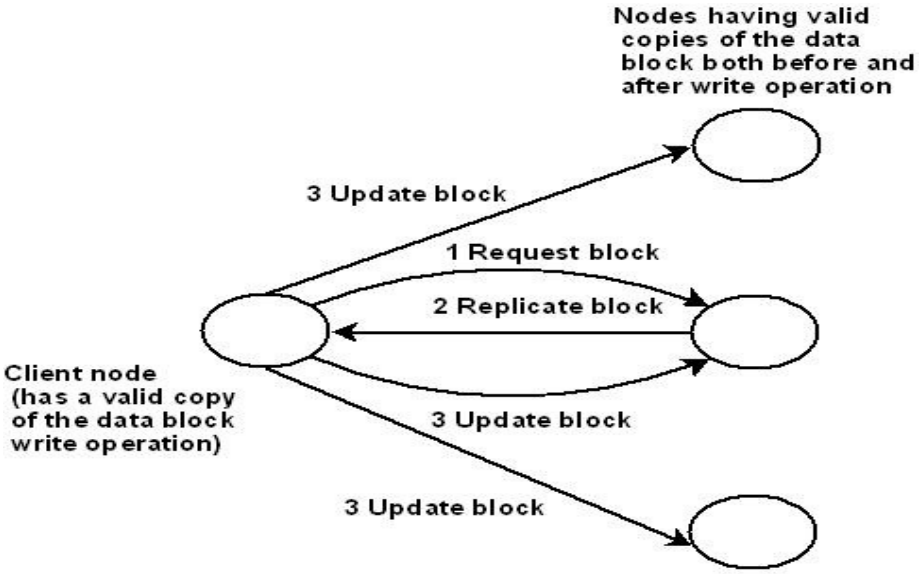


Fig 10. write update memory coherence approach for replicated, migrating blocks strategy

A write operation is carried out by updating all copies of the data on which the write is performed.

When write fault occurs the fault handler copies the accessed block from one of the block's current node to its own node, updates all copies of the block by performing a write operation on the local copy of the block and sending the ***address*** of the modified memory location and its ***value***

The write operation completes only after all the copies of the block have been successfully updated

Sequentially consistency can be achieved by using a mechanism to totally order the write operations of all the node

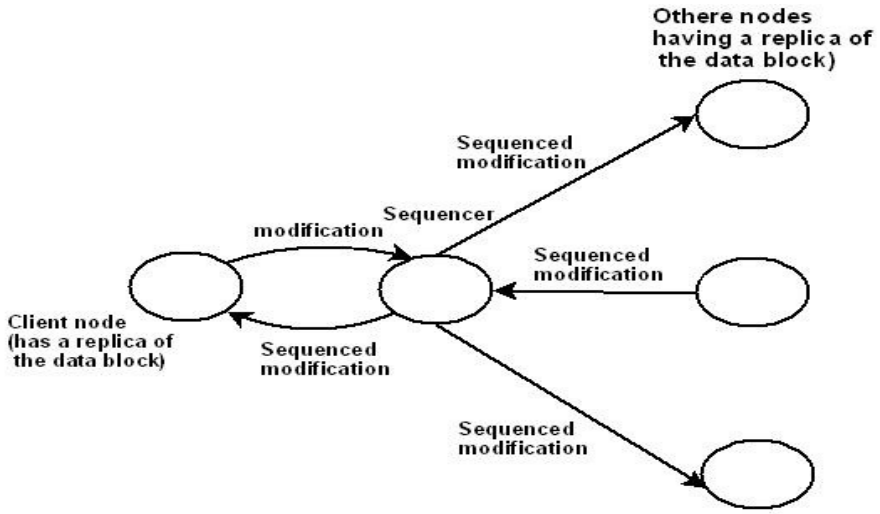


Fig 11. globle sequencing mechanism to sequence the write operations of all nodes

The intended modification of each write operation is first sent to the ***global sequencer***

The intended modification of all the write is first sent to the sequencer

Sequence number to the modification and multicasts the modification with this sequence number to all the nodes where a replica of the data block to be modified is located

The write operations are processed at each node in sequence number order

If the verification fails, node request the sequencer for a retransmission of the missing modification

Write-update approach is very expensive

In write-invalidate approach, updates are only propagated when data are read and several updates can take place before communication is necessary

There is a status tag with each block while write invalidate is implemented

Status tag indicates whether block is valid or shared or read only or writable. With this information the ***read and write request*** are carried out

Read Request

If there is a local block containing the data and if it is valid, the request is satisfied by accessing the local copy of data

Otherwise, the fault handler of the requesting node generates a read fault and obtains the copy of the block from a node having valid copy of the block. If the block is writable in some other block then the retrieved block is made read only.

Write request:

If there is a local block containing the data and if it is valid and writable, the request is immediately satisfied by accessing the local copy of the data

Otherwise, the fault handler of the requesting node generates a write fault and obtain a valid copy of the block and changes its state to writable. A write fault for a block causes the invalidation of all the other copies of the block

DATA LOCATING IN THE RMB STRATEGY

Data-locating issues are involved in the write-invalidate protocol used with the RMB strategy:

1. Locating the owner of a block, the most recent node to have write access to it
2. Keeping track of the node that are currently have a valid copy of the block

Following algorithms may be used:

1. Broadcasting
2. Centralized-server algorithm
3. Fixed distributed-server algorithm
4. Dynamic distributed-server algorithm

Broadcasting

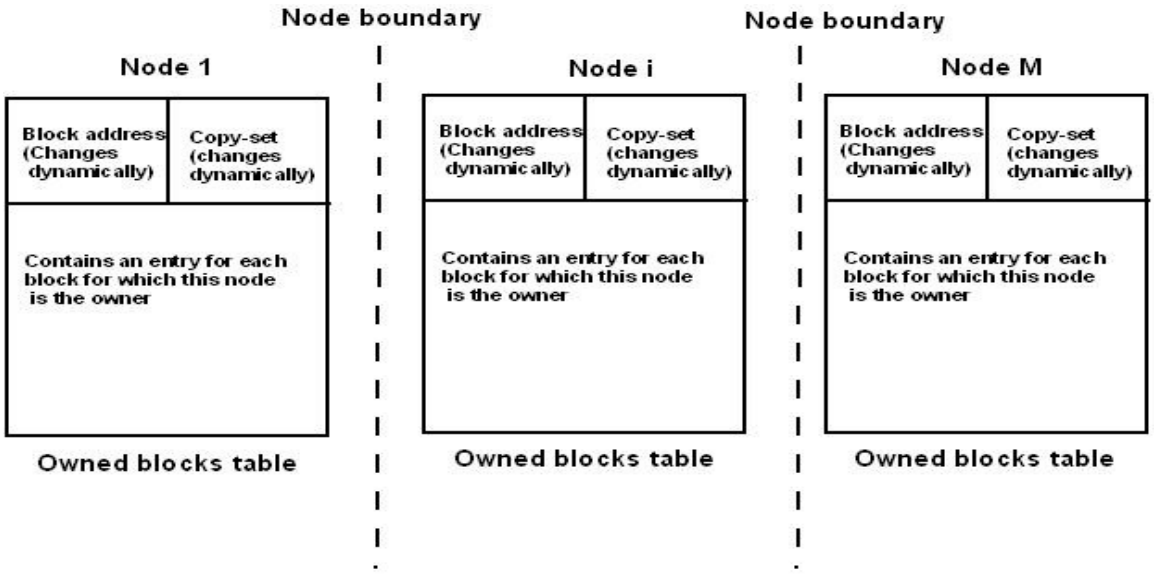


Fig 12 structure and locations of owned blocks table in the broadcasting data locating mechanism for RMB strategy

Each node has an owned block table. This table of a node has an entry for each block for which the node is the owner. Each entry of this table has a copy-set field, contains a valid copy of the corresponding block

When the read fault occurs, the faulting node sends a broadcast read request on the network. When a write fault occurs the node send a broadcast write request on the network . On receiving this request, the owner of the block relinquishes its ownership to node n and sends the block and its copy set to node N. When node N receives the block and the copy set, it sends an invalidation message to all nodes in the copy set. Node N becomes the new owner of the block, an entry is made for the block in its local owned block table. The copy-set field of the entry is initialized to indicate that there are no other copies of the block. Method suffers from disadvantage mentioned during the description of the data-locating mechanism for the NRMB strategy

Centralized-server algorithm

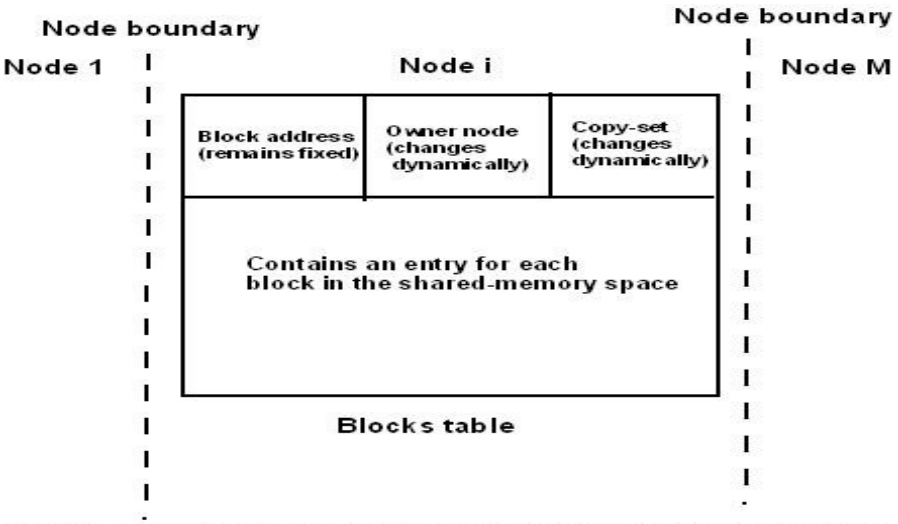


Fig 13. structure and location of block table in the centralized server data locating mechanism for RMB strategy

This is similar to the centralized-server algorithm of the NRMB strategy

Each entry of the block table, managed the centralized server, has an owner-node field

Copy-set field that contains a list of nodes having a valid copy of the block

When read/write fault occurs, the node sends a read/write fault request for the accessed block to the centralized server

For read fault, the centralized server adds node N to the blocks copy set and returns the owner node information to node N

For write fault, it returns both the copy set and owner node information to node N and then initializes the copy set field to contain only node N.

Node N send the request for the block to the owner node. The owner node return a copy of the block to node N

Node N also sends an invalidate message to all nodes in the copy set.

Node N can then perform the read/write operation

Fixed distributed-server algorithm

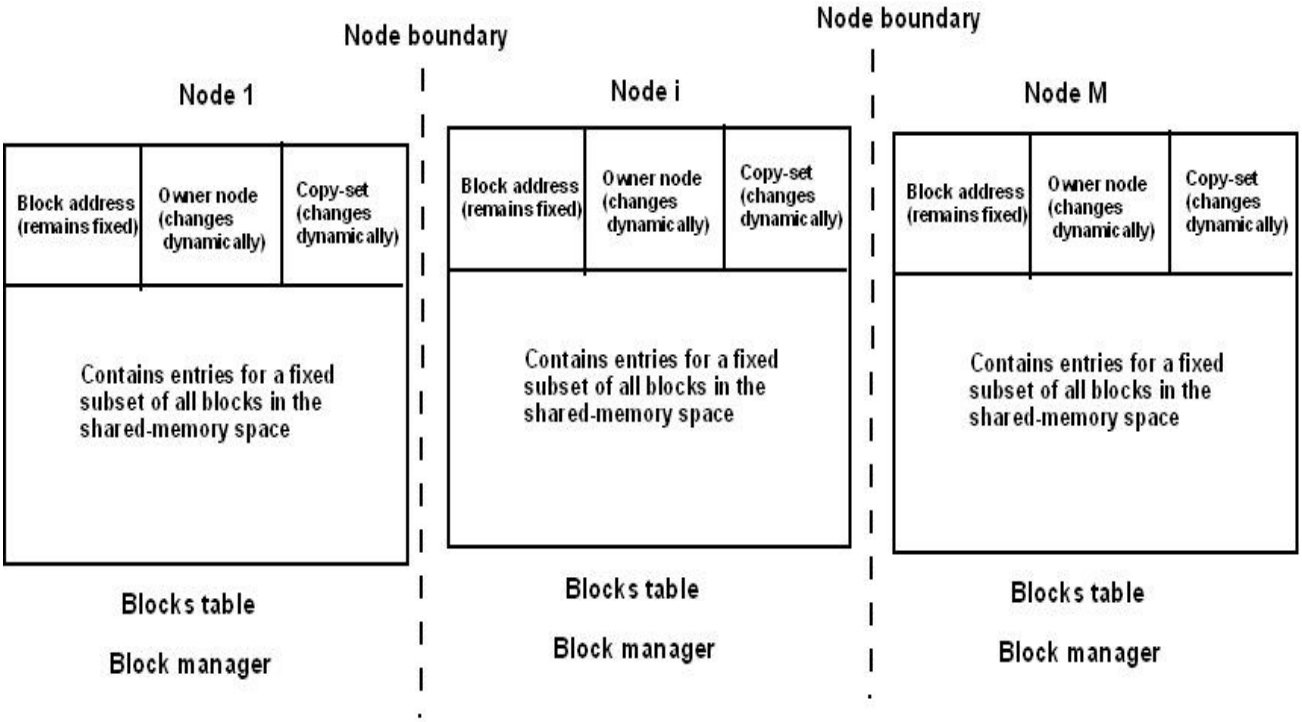


Fig 14 structure and locations of block table in the fixed distributed server data locating mechanism for RMB strategy

This scheme is a direct extension of the centralized-server scheme

Role of the centralized server is distributed to several distributed servers

There is a block manager on several node

Each block manager manages a predetermined subset of block,

Mapping function is used to map a block to a particular block manager.

When fault occurs, the mapping function is used to find the location of the block manager that is managing the currently requested block.

Then the request for the accessed block is send to the block manager of that node.

Advantage is same as the data locating mechanism for the NRMB strategy

Dynamic distributed-server algorithm

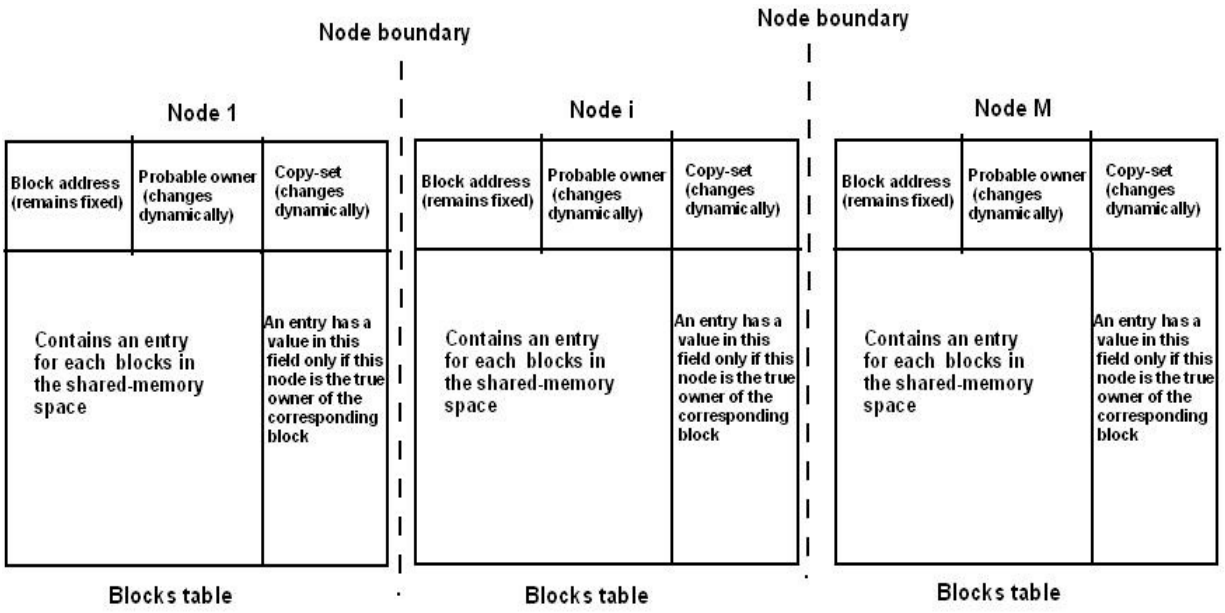


Fig 15. structure and locations of block table in the dynamic distributed server data locating mechanism for RMB strategy

Similar manner as the dynamic distributed server algorithm of the NRMB strategy

Each node has a block table that contains an entry for all block in the shared memory space

Each entry of the table has a probable owner field that gives the node a hint on the location of the owner of the corresponding block.

If the node is the true owner of a block, the entry for the block in the block table of the node also contains a copy set field that provides a list of nodes having a valid copy of the block

When a fault occurs, the fault handler of the faulting node extracts the probable owner node information, send a request for the block to that node

On receiving the block and copy set information, node and sends an in validation request to all nodes in the copy-set.

REPLICATED, NONMIGRATING BLOCK

A shared memory block may be replicated at multiple node of the system but the location of each replica is fixed.

All replicas of a block are kept consistent by updating them all in case of a write excess.

Sequential consistency is ensured by using a global sequencer to sequence the write operation of all nodes.

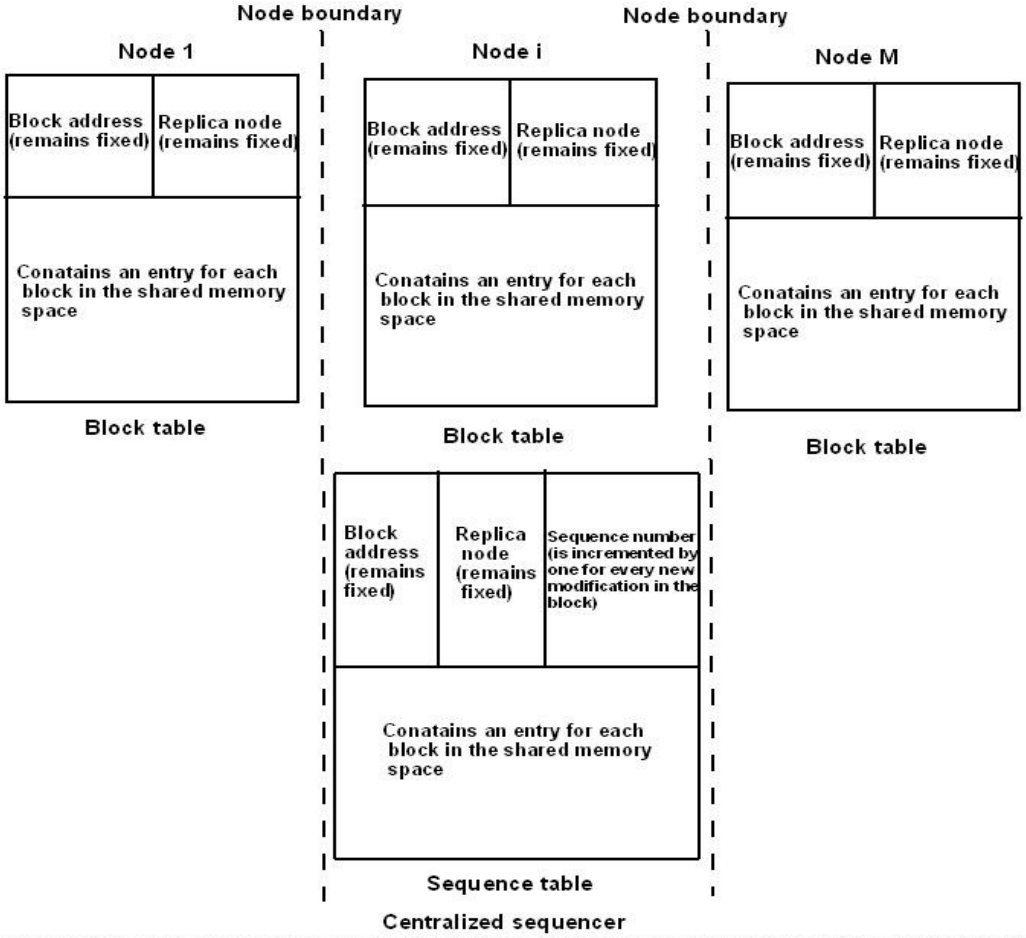
DATA LOCATING IN THE RNMB STRATEGY

Following characteristics:

The replica location of a block never change

All replicas of a data block are kept consistent

Only a read request can be directly send to one of the node having a replica.



structure and location of block table and sequence table in the centralized sequencer data locating mechanism for RNMB strategy

The block table of a node has an entry for each block in the shared memory.

Each entry maps to one of the block to its replica locations

The sequence table also has an entry for each block in the shared memory space.

It contains three field

- block address
- Replica set field(list of nodes having replica)
- Sequence No. (incremented by 1 for every modification)

Read operation directly done between the nodes by referring to the block table

A write operation on a block is sent to sequencer

The sequencer assign the next sequence number to the requested modification

It then multicast the modification with this sequence number to all the nodes listed in the replica set field.

Munin: A RELEASE CONSISTENT DSM SYSTEM

Structure of Shared-Memory Space

The shared memory space of Munin is structured as a collection of shared variables.

The shared variables are declared with the keywords shared so the compiler can recognized them.

The programmer can annotate the shared variable with any of the annotation type.

Each shared variable, by default, is placed by the compiler on a separate page.

Unit of data transfer across the network by MMU hardware.

Multiple shared variables having the same annotation types can be placed in the same page

Different annotation types cannot be placed in the same page

Variables of size larger than the size of the page occupies multiple pages

Implementation of Release Consistency

Release consistency application must be modeled around critical sections.

Munin provides two such ***synchronization mechanism***.

A locking mechanism

A barrier mechanism.

The locking mechanism uses lock synchronization variables with ***acquirelock*** and ***releasedlock*** primitives for accessing these variables.

The acquire lock primitive with a lock variable as its parameter is executed by a process to enter a critical section .

Release lock primitive with the same lock variable parameter is executed by the process to exit from the critical section.

When a process makes an acquire lock request the system first checks if the lock variable on the local node.

If not, the probable owner mechanism is used to find the location owner of the lock variable and request is sent to that node.

If the lock is free it is granted to the requesting process

Otherwise the requesting process is added to the end of the queue waiting to acquire the lock variable.

When the lock variable is released it is given to the next process waiting in the queue.

The barrier mechanism uses barrier synchronization variables with a ***waitAtBarrier*** primitive for accessing these variables.

Barriers are implemented by using the ***centralized barrier server mechanism***.

In a network fault occur, the ***probable owner based dynamic distributed server algorithm*** is used in Munin to locate a page containing the accessed shared variables.

It is similar to the copy set mechanism used to keep track of all the replicas location

Annotation for shared variables

The release consistency of Munin allow applications to have better performance than sequentially consistent DSM system.

For further performance improvements, Munin defines several standard annotation for shared variables.

The standard annotations and consistency protocol for variables for variable for each type are as follows:

1. *Read-only*
2. *Migratory*
3. *Write-shared*
4. *Producer-consumer*
5. *Result*
6. *Reduction*
7. *Conventional*

Read-only

Shared-data variables annotated as read-only are immutable data items

These variable are read but never written after initialization.

No consistency problem.

They can be freely replicated in all nodes

Read-only variables are protected by the MMU hardware

An attempt to write on a read only data will create a fatal error

Migratory

Shared variable that are accessed in phases, where each phase correspondence to a series of accesses by a single process, may be annotated as migratory variables

The locking mechanism is used to keep migratory variable consistent

To access a migratory variable first acquires a lock for the variable uses for some time and then release the lock after finishing

Page migrate from one node to another on a demand basis, but page are not replicated

Only one copy of a page containing a migrating variable exists in the system

The lock and the message are sent together as a single message to the location of the next process that is given the lock for accessing it

Write-shared

A programmer may use this annotation with a shared variable to indicate to the system that the variable is updated concurrently by multiple processes

For example in a matrix the values of the rows and columns can be updated by different process

Munin avoid the false sharing problem of write shared variable

It allows to update them concurrently

Write shared variable is replicated on all node

When access to the write shared variables causes a network page fault to occur.

The page having the variables copies it to the faulting node

If the access is a write access the system first make a copy of the page(called the ***twin page***)

The process may perform several write to the page before releasing it

When page is released, the system perform a word by word comparison of the original page with the twin page and sends the differences to all the nodes having the replica of the page.

When a node receives the differences of a modified page, the system check if the local copy of the page was modified

Producer-consumer

Shared variable that ***written (producer) by only one process*** and ***read(consumed) by fixed set*** of other processes may be annotated to be of producer-consumer type

Munin uses an " ***eager object movement***" mechanism

In this mechanism a variable is moved to the from the producers node to the consumer node in advance avoiding the network page fault

Write update protocol is used when ever the producer updates the variable.

Producer may send several update together by using ***locking mechanism***

In this case procedure acquires a synchronization lock, make several update on the variable and then releases the lock

Result

Result variable are just the opposite of producer- consumer variable

They are ***written by multiple processes*** but ***read by only one process***

Different process write to different parts of the variables that do not conflict

The variable is read only when all its parts have been written

Example in matrix ***worker process*** would generate and fill the elements of each row and column

Once matrix is complete it is used by the ***master process***

Munin uses a special ***write-update protocol*** for result variable in which updates are sent to the nodes having master process and not all replica location

Reduction

Shared variable that must be atomically modified may be annotated to be of reduction type

Example: parallel computation application, a global minimum must be atomically fetched and modified

For better performance a reduction variable is stored at a ***fixed owner*** that receives updates to the variable from other process synchronize the updates , performs the updates on its variable and propagates the updated variable to all the replica location

Conventional

Shared variable that are not annotated as one of the above types are conventional variables

Page containing a conventional variable is dynamically moved to the location of a process that want to perform a write operation on the variable

Uses ***Write Invalidate***

Read only, migratory and write shared annotation types are useful and frequently used

Producer consumer, result and reduction annotation types are less frequently used

REPLACEMENT STRATEGY

In DSM system that allow shared memory block to be dynamically migrated/replicated

Following issue:

1. Which block should be replaced to make space for a newly required block?
2. Where should the replaced block be placed?

Which block to replace

Classification of replacement algorithms:

1. Usage based verses non-usage based
2. Fixed space verses variable space

Usage based verses non-usage based

Uses based algorithms keep track of the history of usage of a cache line and use this information to make replacement decisions eg. LRU algorithm

Non-usage-based algorithms do not take the record of use of cache lines into account when doing replacement. First in first out and Random (random or pseudorandom) belong to this class

Fixed space versus variable space

Fixed-space algorithms assume that the cache size is fixed while variable space algorithm are based on the assumption that the cache size can be changed dynamically depending on the need

In a variable space algorithm, a fetch does not imply a replacement, and a swap-out can take place without a corresponding fetch

Variable space algorithms are not suitable for a DSM system

- Usage based and non-usage based algorithm are suitable for DSM system

In DSM system of IVY, each memory block of a node is classified into one of the following five types:

Unused: a free memory block that is not currently being used

Nil: a block that has been invalidated

Read-only: a block for which the node has only read access right

Read-owned: a block for which the node has only read access right but is also the owner of the block

Writable: a block for which the node has write access permission

Based on this classification of block, ***priority*** is used:

1. Both unused and nil block have the highest replacement priority
2. The read-only block have the next replacement priority because a copy of the data will be present with owner node
3. Read-owned and writable block for which replica(s) exist on some other node(s) have the next replacement priority
4. Read-owned and writable block for which only this node has a copy have the lowest replacement priority

An LRU policy is used to select a block for replacement when all the blocks in the local cache have the same priority

Where to place a replaced block

Once a memory block has been selected for replacement, it should be ensured that if there is some useful information in the block , it should not be lost.

The two commonly used approaches for storing a useful block as follow:

1. ***Using secondary store:***

The block is simply transferred on to a local disc.

Advantages: it does not waste any memory space

1. ***Using the memory space of other nodes:***

It may be faster to transfer a block over the network than to transfer it to a local disc.

Methods require each node to maintain a table of free memory space in all other nodes.

Thrashing

Thrashing is said to occur when the system spends a large amount of time transferring shared data blocks from one node to another

Thrashing may occur in following situation:

1. When interleaved data accesses made by processes on two or more nodes
2. When blocks with read only permissions are repeatedly invalidated soon after they are replicated

Thrashing degrades system performance considerably

Methods for solving Thrashing problems:

1. ***Providing application controlled locks.*** Locking data to prevent other node from accessing that data for a short period of time can reduce Thrashing.
2. ***Nailing a block to a node for a minimum amount of time***

Disallow a block to a be taken away from a node until a minimum amount of time t elapses after its allocation to that node.

The time t can be either fixed statically or be tuned dynamically on the basis of access patterns.

It is also decided by the length of the queue waiting for it to access

Drawback:

It is very difficult to choose the appropriate value for the time.

Time t may elapse before all the write operation gets completed.

3. ***Tailoring the coherence algorithm to the shared data usage patterns***

Thrashing can be minimized by using different coherence protocol for shared data having different characteristics.

SUBJECT TITLE: DISTRIBUTED COMPUTING
SUBJECT CODE : SCSX1028
UNIT IV
SYNCHRONIZATION AND MANAGEMENT

SYNCHRONIZATION

- A distributed system consists of a collection of distinct processes that are spatially separated and run concurrently.
- In system with concurrent multiple processes will share their resources.
- Therefore activity of one process with the resource would influence the other process.
- For example a resource can not be used simultaneously by multiple processes, a process willing to use must wait until the next process finish.
- In file access operation the client process and the file server process must co operate.
- The rules enforced for correct interaction are implemented in the form of synchronization mechanism
- Synchronization mechanisms that are suitable for distributed systems. In particular, the following synchronization

Related issues are described:

1. Clock synchronization
2. Event ordering
3. Mutual exclusion
4. Deadlock
5. Election algorithm

CLOCK SYNCHRONIZATION

- Every computer needs a timer mechanism called a computer clock
- To keep track of current time and also for various accounting purposes such as calculating the time spent by a process in CPU utilization, disk I/O and so on, so that the corresponding user can be charged properly.
- In a distributed system, an application may have processes that concurrently run on multiple nodes of the system.
- For correct results, several such distributed applications require that the clocks of the nodes are synchronized with each other.
- In a distributed system, synchronized clocks also enable one to measure the duration of distributed activities that start on one node and terminate on another node
- For instance calculating the time taken to transmit a message from one node to another at any arbitrary time.

- It is difficult to get the correct result in the case if the clocks of the sender and receiver nodes are not synchronized.
- It is the job of a ***distributed operating system designer*** to devise and use suitable algorithms for properly synchronizing the clocks of a distributed system.

HOW COMPUTER CLOCKS ARE IMPLEMENTED

A computer clock usually consists of three components

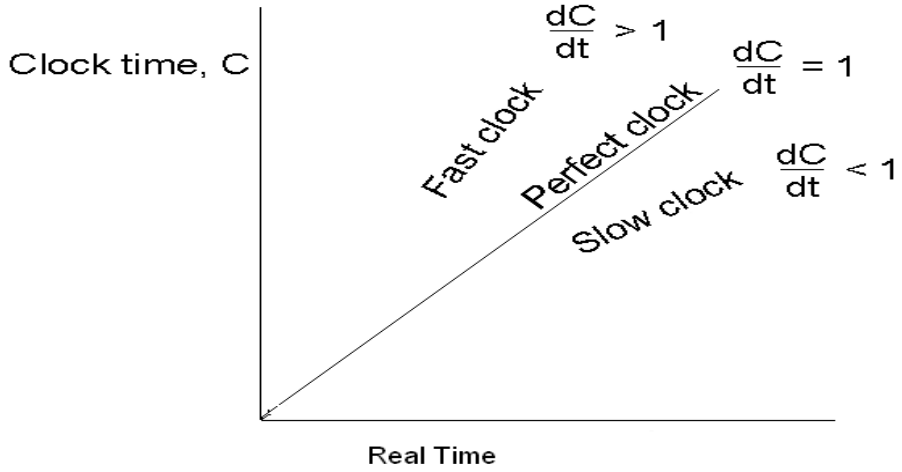
1. A quartz crystal that oscillates at a well – defined frequency
2. A counter register- the value in the counter register is decremented by 1 for each oscillation of the quartz crystal
3. A holding register - to store a constant value that is decided based on the frequency of oscillation of the quartz crystal

- When the value of the counter register becomes zero, an ***interrupt*** is generated and its value is reinitialized to the value in the holding register.
- Each interrupt is called ***clock tick***.
- In order to make the computer clock function as a ordinary clock the following things are done
- The value in the holding register is chosen ***60*** so that on 60 clock ticks occur in a second.
- The computer clock is synchronized with real time (external clock)

DRIFTING OF CLOCKS

- A clock always runs at a constant rate because its quartz crystal oscillates at a well-defined frequency
- Due to difference in the crystals the rates at which two clocks run are normally different from each other.
- The difference in the oscillation period between two clocks might be extremely small but the difference accumulated over many oscillations leads to an observable difference in the times of the two clocks
- The drift time is approx. $10^{-6,}$ a difference of 1 sec. Every 1, 00,000 sec.
- A computer clock must be periodically re synchronized with real time clock to keep it non faulty
- A clock is considered non faulty if there is a bound on the amount of drift from real time for any given finite time interval.
- Let us suppose that when the real time is t , the time value of a clock p is Cp(t).
- If all clock in the world were perfectly synchronized, Cp(t)=t for all p and t.
- If C denotes the time value of a clock, dC/dt should be 1.
- Maximum drift rate allowable in p,a clock is said to be non-faulty if the following condition hold for it :$1-\rho <= dC/dt <= 1+\rho$

- After synchronization with perfect clock, slow and fast clocks drift in opposite directions from the perfect clock.



- A distributed system consists of several nodes, each with its clock, running at its own speed.
- Because of the nonzero drift rates of all clocks, the set of clocks of a distributed system do not remain well synchronized without some periodic resynchronization.
- Distributed system must periodically resynchronize their local clocks to maintain a global time base across the entire system.
- A distributed system requires the following types of clock synchronization

SYNCHRONIZATION OF THE COMPUTER CLOCKS WITH REAL TIME(OR EXTERNAL) CLOCKS

- This type of synchronization is mainly for real time applications.
- External clock synchronization allows the system to exchange information about the timing of events with other systems and users.
- Coordinated universal time (UTC) is an international standard.
- Is a external time source used in synchronizing computer clocks with real time.
- Many standard bodies disseminate UTC signals by radio telephone and satellite.
- Computers equipped with time provided devices can synchronize their clocks with timing signals.

MUTUAL (OR INTERNAL) SYCHRONIZATION OF THE CLOCKS IN DIFFERENT NODES OF SYSTEM

- This type of synchronization is mainly required for those Applications that requires
- A consistent view of time across all nodes of a distributed system

- For the measurement of the duration of the distributed activities that terminate on a node different from the one on which they start

CLOCK SYNCHRONIZATION ISSUES

- No two clocks can be perfectly synchronized.
- Two clocks are said to be synchronized at particular instant of time if the difference in the time value of the two clock is less some specified constant δ.
- The difference in time value of two clocks is called clock skew.
- The set of clocks are said to be synchronized if the clock skew of any two clocks in this set is less than delta δ.
- Clock synchronization requires each node to read other nodes clock values.
- The mechanism used by a node to read other clocks differ from one algorithm to another.
- A node can at time only an approx. View of its clock skew with respect to other nodes clocks in the system.
- Error occurs mainly because of unpredictable communication during message passing used to deliver a clock signal or a clock message from one node to another
- A minimum value of the unpredictable communication delays between two nodes can be computed by counting the time needed to prepare, transmit and receive an empty message in the absence of transmission errors and any other system load.
- It is impossible to calculate the upper bound of this value because it depends computation going on the amount of communication and computational growing on in parallel in the system, on the possibility the transmission errors will cause messages to be transmitted several times.

ISSUE

- In clock synchronization the time must never ru backward because this could cause serious problems such as the repetition of certain operation that may be hazardous in certain cases.
- During synchronization the fast clock has to be slowed down if the time of the fast clock is re adjusted 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 amt of time to be added to the clock time for each interrupt.

CLOCK SYNCHONIZATION ALGORITHMS

Broadly classified as

Centralized

- Passive Time server centralized algorithm
- Active Time server centralized algorithm

Distributed

- Global Averaging distributed algorithm
- Localized Averaging distributed algorithm

CENTRALIZED ALGORITHM

- One node has a real time receiver called as time server node.
- Clock time of this node is used as the reference time.
- The clocks of all other nodes synchronized with the clock time of the time server node

Passive Time server centralized algorithm

- Each node periodically send a message ***time=? t***o the time server
- When the time server receives the message, it quickly respond with the message ***time=T*** where T is the current time of time server.
- T_o is time when ***time=?*** Message is generated, T_1 is the time when ***time=T*** is received.
- The propagation of message from the time server node to the client node is $(T_o - T_1)/2$
- The clock is readjusted to $T + (T_o - T_1)/2$
- $(T_o - T_1)/2$ is not a good estimate

Two methods to improve this estimated values

Method 1

- Approx. Time taken by the time server to handle the interrupt is taken as ***I.***
- The propagation of message from the time s server node to the client node is (T_o -T_1 -I)/2
- The clock is readjusted to T +(T_o -T_1 -I)/2

Method 2

- Several measurements of T_o -T_1
- Considering a threshold, the value T_o -T_1 exceeds this threshold is discarded.
- The average of the remaining measurements are found
- Half of this value is used as the value to be added to T

Active Time server centralized algorithm

- The active time server periodically broadcasts its clock time time=T to all nodes.
- Each node has the prior knowledge of the approx. Time T_a required for the propagation of message.
- The nodes clock gets readjusted to the time T+ T_a .

Drawbacks

1. If the broadcast message reaches too late the node gets readjusted to an in correct value.

2. Broadcast facility has to be supported by the network

Overcomes the draw back(Berkeley Alg.)

- The time server periodically send message time=? To all nodes.
- Each node send back its clock value.
- The time server has knowledge of approx. propagation time.
- Based on this knowledge it readjust the clock values of the reply message.

DRAW BACKS OF CENTRALIZED ALGORITHM

1. Single point failure

2. Scalability

DISTRIBUTED ALGORITHM

- Internally synchronized.
- Each node has a real time receiver so it gets independently synchronized with real time.
- Multiple real time clocks

GLOBAL AVERAGING DISTRIBUTED ALGORITHM

- Each node broad casts its clock time in the form of ***resync***message
- Local time=T_O***+iR***
- ***i= some integer***
- T_O ***= fixed time in the past agreed upon by all nodes.***
- ***R= S/m parameter that depends on factors, total no. Of nodes, max. Allowable drift rate etc.,***
- ***Resyn*** message is broad cast from each node at the beginning of every fixed length resynchronization interval.
- Clocks run differently in different nodes therefore broadcast does not happen simultaneously

---

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.
- 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.

An algorithm for implementing mutual exclusion must satisfy the following requirements.

Mutual Exclusion: Shared resource accessed by multiple concurrent processes at any time only one process should access the resource.

A process that has been granted the resource must release it before it can be granted to another process.

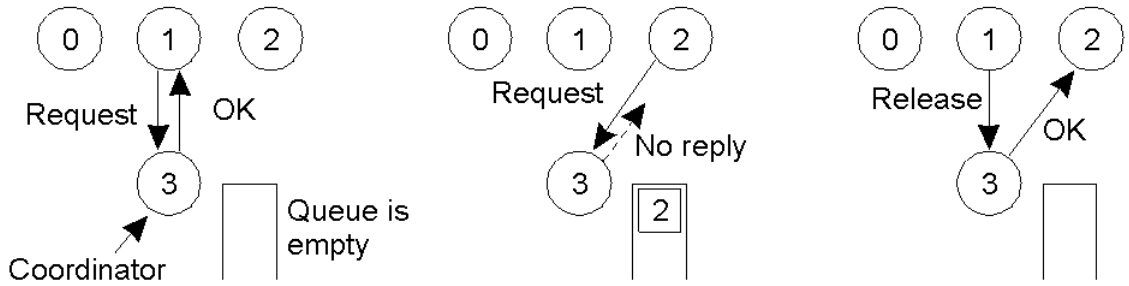
No starvation: Ifevery process that is granted the resource eventually released it, every request must be eventually granted

Algorithms for implementing mutual exclusion

CENTRALIZED APPROACH:

- One of the processes in the system is elected as the coordinator and coordinates the entry to the critical section.

- Each process that wants to enter a centralized server must first seek permission from the coordinator.
- If no other process is currently in that centralized server the coordinator can immediately grant permission to the requesting process.
- If 2 or more processes concurrently ask for permission to enter the same centralized server, the coordinator grants permission to only one process at a time in accordance with some scheduling algorithm
- After executing a centralized server, when a process exit centralized server, it must notify the coordinator so that the coordinator can grant permission to another process.



(a) (b) (c)

- This algorithm ensure mutual exclusion, at a time the coordinator allows only one process to enter centralized server.
- Algorithm also ensure that no starvation will occur because of the use of FCFS scheduling policy
- Simple to implement and requires only 3 message per centralized server entry

Drawback:

- Single coordinator is subjected to single point failure

Distributed Approach

- The decision making for mutual exclusion is distributed across the entire system.
- All processes that want to enter the same centralized server cooperate with each other before reaching a decision on which process will enter the centralized server next.
- When a process wants to enter a centralized server, it sends a request message to all other processes.

The message contain the following information

1. The process identifier of the process

2. The name of the centralized server that the process wants to enter
3. A unique timestamp generated by the process for the request message.

- On receiving a request message, a process either immediately sends back a reply message to the sender or defers sending a reply based on the following.
- If the receiver process is itself currently executing in the centralized server, it simply queues the request message and defers sending a reply.
- If the receiver process is currently not executing in the centralized server but is waiting for its turn ,it compares the timestamp in the received request message with the timestamp in its own request message that it send to other process.
- If the timestamp of the received request message is lower, it means that the sender process made a request before the receiver process to enter the centralized server. The receiver process immediately sends a reply message to the sender. If the receiver process own request message has a lower timestamp, the receiver queues the received request message and defers sending a reply message.
- If the receiver process neither is in centralized server nor is waiting for its turn to enter the centralized server, it immediately sends back a reply message.
- A process that sends out a request message keeps waiting for reply message from other processes.
- It enters the centralized server as soon as it has received reply message from all processes.
- After if finishes executing in the centralized server, it sends reply message for all processes in its queue and deletes them from its queue.

Drawbacks

- If one process fails, entire scheme collapses, because the failed process will not reply to the request message, causing all the requesting processes to wait indefinitely.
- Requires each process know the identity of all the processes participating in the mutual exclusion algorithm. Each process of a group needs to dynamically keep track of the processes entering or leaving the group.
- Process enter in centralized server can do only after communicating with all other processes and getting permission from them.
- Token passing approach
- Mutual Exclusion is achieved by using a single token that is circulated among the processes.
- A ***token*** is a special type of message that entitles its holder to enter a cs.
- 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 in the same direction.

The algorithm requires the handling of failures:

1. ***Process failure:***

- A process failure in the system causes the logical ring to break.
- A new logical ring must be established to ensure the continued circulation of the token among other processes
- Detection of a failed process can be easily done by making it a rule that a process receiving the token from its neighbor sends an ack. Message to its neighbor within a fixed timeout period.
- Dynamic reconfiguration of the logical ring can be done.
- When a process detects that its neighbor has failed, it remove the failed process by skipping it and passing the token to the process after it.
- When a process become alive after recovery ,it informs the neighbor previous to it in the ring, to get the token in next circulation

2. ***Lost token:***

- If the token is lost, a new token must be generated.
- One of the process on the ring as a ***monitor*** process.
- The ***monitor*** periodically circulate a “who has the token” message on the ring.
- All processes pass this message to their neighbor, except the process that has the token.
- This process writes its id in special field of the message before passing it to its neighbor.
- When the message return to the monitor, it checks the special field of the message.
- If there is no entry, it concludes that the token lost, generates a new token.

Problems:

1. Monitor process itself fail
2. “Who has the token?” message itself get lost.

- Solved by using more than one monitor processes.
- Each monitor process checks the availability of token on the ring.
- When a monitor process detects that the token is lost, it holds an election with other monitor processes to decide which monitor process will generate and circulate anew token.
- An election prevent the generation of multiple tokens.

ELECTION ALGORITHM

Election algorithm are meant for electing a coordinator process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system.

1. Each process in the system has a unique priority no.
2. Whenever an election is held, the process having the highest priority no among the currently active processes is elected as the coordinator.
3. On recovery, a failed process can take appropriate actions to rejoin the set of active processes.

Bully Algorithm

When any process notices that the coordinator is no longer responding to the requests, it asks for the election.

Example: A process P holds an election as follows

1) P sends an ELECTION message to all the processes with higher numbers.

2) If no one responds, P wins the election and becomes the coordinator.

3) If one higher process answers; it takes over the job and P's job is done.

At any moment an "election" message can arrive to process from one of its lowered numbered colleague. The receiving process replies with an OK to say that it is alive and can take over as a coordinator. Now this receiver holds an election and in the end all the processes give except one and that one is the new coordinator.

The new coordinator announces its new post by sending all the processes a message that it is starting immediately and is the new coordinator of the system. If the old coordinator was down and if it gets up again; it holds for an election which works in the above mentioned fashion. The biggest numbered process always wins and hence the name "bully" is used for this algorithm.

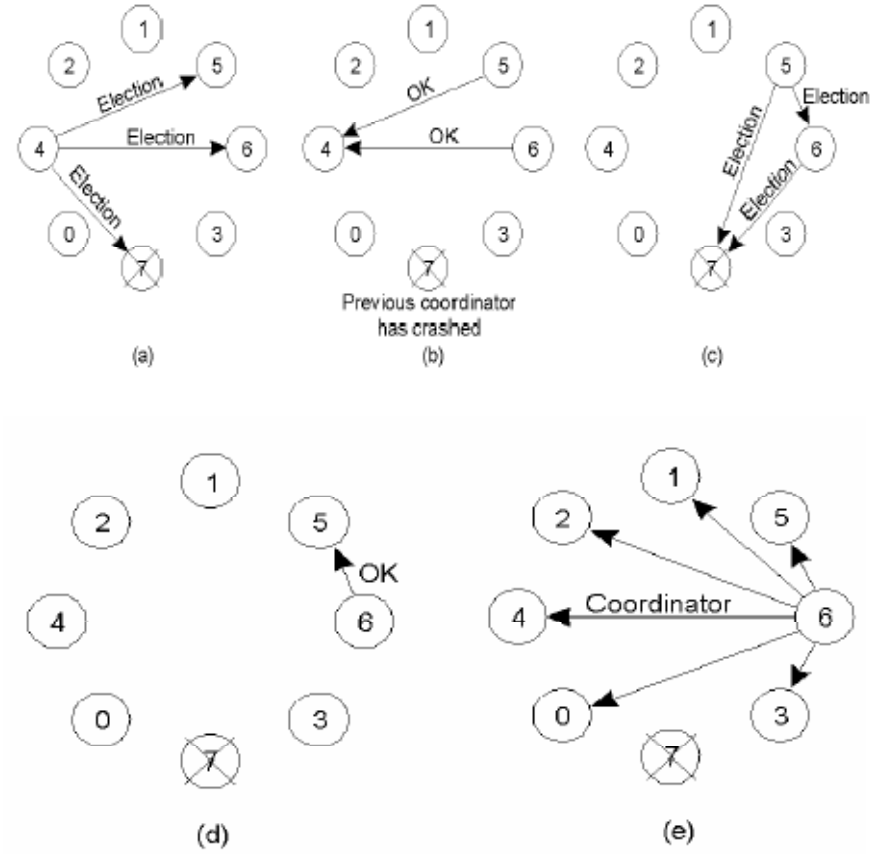


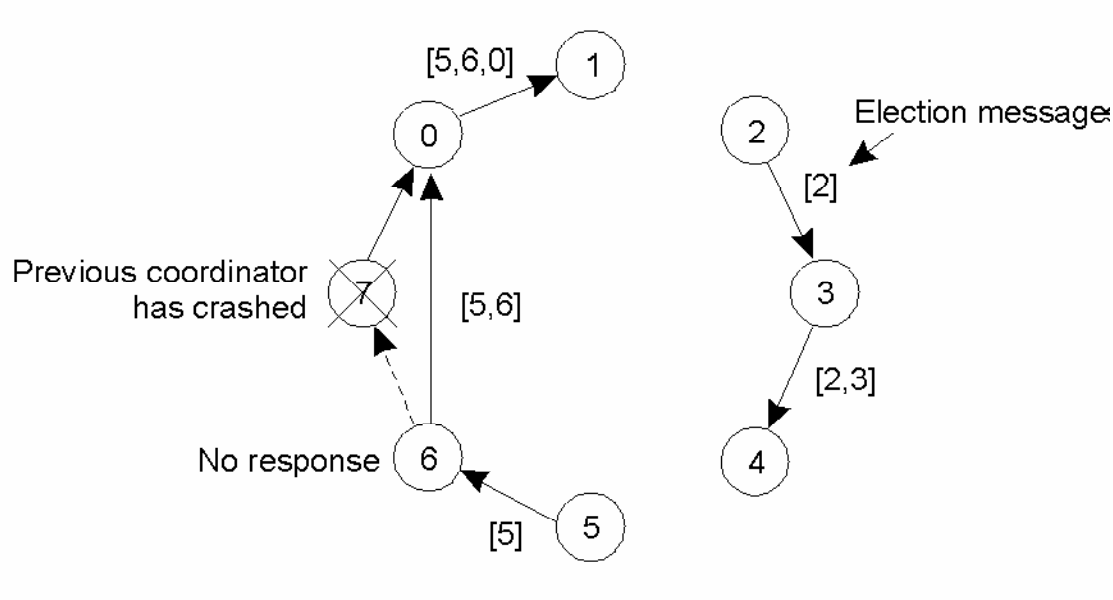
Figure 6.1: The bully election algorithm

a) **Process 4 holds an election.**
b) **Process 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.**

Ring Algorithm:

- It is based on the use of a ring as the name suggests. But this doesnot 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 meaage 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.



Election algorithm using Ring

DEADLOCKS

- In a multiprogramming system, processes request resources.
- If those resources are being used by other processes then the process enters a waiting state.
- If other processes are also in a waiting state, we have deadlock.

Definition: A set of processes is in a deadlock state if every process in the set is waiting for an event (release) that can only be caused by some other process in the same set.

Example
Process-1 requests the printer, gets it
Process-2 requests the tape unit, gets it Process-1 and
Process-1 requests the tape unit, waits Process-2 are
Process-2 requests the printer, waits deadlocked!

We analyze deadlocks with the following assumptions:

- A process must request a resource before using it. It must release the resource after using it. (request use release)
- A process cannot request a number more than the total number of resources available in the system.
- For the resources of the system, a resource table shall be kept, which shows whether each process is free or if occupied, by which process it is occupied.
- For every resource, queues shall be kept, indicating the names of processes waiting for that resource.
- A deadlock occurs if and only if the following four conditions hold in a system simultaneously:

1. Mutual Exclusion:

- At least one of the resources is non-sharable that is only a limited number of processes can use it at a time and if it is requested by a process while it is being used by another one, the requesting process has to wait until the resource is released.

2. Hold and Wait:

- There must be at least one process that is holding at least one resource and waiting for other resources that are being hold by other processes.

3. No Preemption:

- No resource can be preempted before the holding process completes its task with that resource.

4. Circular Wait:

There exists a set of processes: {P1, P2, ..., Pn} such that
P1 is waiting for a resource held by P2
P2 is waiting for a resource held by P3
...
Pn-1 is waiting for a resource held by Pn
Pn is waiting for a resource held by P1

Methods for handling deadlocks are:

• Deadlock prevention
• Deadlock avoidance
• Deadlock detection and recovery.

Deadlock Prevention

- To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be discarded.

The methods for those conditions are as follows:

Mutual Exclusion:

- We do not have systems with all resources being sharable. Some resources like printers, processing units are non-sharable.
- So it is not possible to prevent deadlocks by denying mutual exclusion.

Hold and Wait:

- One protocol to ensure that hold-and-wait condition never occurs says each process must request and get all of its resources before it begins execution.
- Another protocol is “Each process can request resources only when it does not occupies any resources.”
- The second protocol is better.
- Both protocols cause low resource utilization and starvation.
- Many resources are allocated but most of them are unused for a long period of time.
- A process that requests several commonly used resources causes many others to wait indefinitely.

No Preemption:

- One protocol is “If a process that is holding some resources requests another resource and that resource cannot be allocated to it, then it must release all resources that are currently allocated to it.”
- Another protocol is “When a process requests some resources, if they are available, allocate them.
- If a resource it requested is not available, then we check whether it is being used or it is allocated to some other process waiting for other resources.
- If that resource is not being used, then the OS preempts it from the waiting process and allocate it to the requesting process.
- If that resource is used, the requesting process must wait.”
- This protocol can be applied to resources whose states can easily be saved and restored (registers, memory space).
- It cannot be applied to resources like printers.

Circular Wait:

- One protocol to ensure that the circular wait condition never holds is “Impose a linear ordering of all resource types.”
- Each process can only request resources in an increasing order of priority.

Deadlock avoidance

- Given some additional information on how each process will request resources, it is possible to construct an algorithm that will avoid deadlock states.
- The algorithm will dynamically examine the resource allocation operations to ensure that there won't be a circular wait on resources.
- When a process requests a resource that is already available, the system must decide whether that resource can immediately be allocated or not.
- The resource is immediately allocated only if it leaves the system in a *safe state*.
- A state is safe if the system can allocate resources to each process in some order avoiding a deadlock.
- A deadlock state is an unsafe state.

Deadlock Detection

- If a system has no deadlock prevention and no deadlock avoidance scheme, then it needs a deadlock detection scheme with recovery from deadlock capability.
- For this, information should be kept on the allocation of resources to processes, and on outstanding allocation requests.
- Then, an algorithm is needed which will determine whether the system has entered a deadlock state.
- This algorithm must be invoked periodically.

TASK ASSIGNMENT APPROACH

The Basic Idea:

- In this approach, a process is considered to be composed ofmultiple tasks and the goal is to find an optimal assignment policyfor the tasks of an individual process.

Typical assumptions found intask assignment work are as follows :

- A process has already been split into pieces called tasks.
- This split occurs along natural boundaries, so that each taskwill have integrity in itself and data transfers among the taskswill be minimized.
- The amount of computation required by each task and thespeed of each processor are known.
- The cost of processing each task on every node of thesystem is known.
- This cost is usually derived based on theinformation about the speed of each processor and theamount of computation required by each task.
- The Interprocesses Communication (IPC) costs betweenevery pair of tasks is known.
- The IPC cost of consideredzero (negligible) for tasks assigned to the same node.
- Theyare usually estimated by an analysis of the static program ofa process.

- For example during the execution of the process,if two tasks communicate *n* times and average time for eachintertask communication is *t,* the intertask communicationcost for the two tasks is *nt*.
- Other constraints, such as resource requirements of thetasks and the available resources at each node, precedencerelationships among the tasks, and so on, are also known.
- Reassignment of the tasks is generally not possible.
- With these assumptions, the task assignment algorithmsseek to assign the tasks of a process to the nodes of the distributedsystem in such a manner so as to achieve goals such as thefollowing.

1. Minimization of IPC costs
2. Quick turnaround time for the complete process
3. A high degree of parallelism
4. Efficient utilization of system resources in general

LOAD BALANCING APPROACH :

- The scheduling algorithms using this approach are known asload balancing algorithms or load leveling algorithms.
- Thesealgorithms are based on the intuition that, for better resourceutilization.
- It is desirable for the load in a distributed system to bebalanced evenly.
- Thus, a load balancing algorithm tries to balancethe total system load by transparently transferring the workloadfrom heavily loaded nodes to lightly nodes in an attempt to ensuregood overall performance relative to some specific metric of systemperformance.
- When considering performance from the user pointof view, the metric involved is often the response time of theprocesses.
- However, when performance is considered from theresources point of view, the metric involved is the total systemsthroughput.
- In contrast to response time, throughput is concernedwith seeing that all users are treated fairly and that all are makingprogress.
- The resource view of maximizing resourceutilization is compatible with the desire to maximize system throughput.
- Thus the basic goal of almost all the load balancingalgorithms is to maximize the total system throughput.

LOAD SHARING APPROACH

- Several researchers believe that load balancing with itsimplication of attempting to equalize workload on all the nodes of the system, is not an appropriate objective.
- This is because theoverhead involved in gathering state information to achieve thisobjective is normally very large, especially in distributed systemshaving a large number of nodes.

- Moreover, load balancing in thesense is not achievable because the number of processes in anode is always fluctuating and the temporal unbalance among thenodes exists at every moment, even if the static (average) load isperfectly balanced for the proper utilization the resources of adistributed system, it is not required to balance the load on all thenodes.
- It is necessary and sufficient to prevent the nodesfrom being idle while some other nodes have more than twoprocesses.
- Therefore this rectification is often called dynamic loadsharing instead of dynamic load balancing.

Issues in Designing Load Sharing Algorithms :

- Similar to the load balancing algorithms, the design of a loadsharing algorithm also requires that proper decisions be maderegarding load estimation policy, process transfer policy, stateinformation exchange policy, location policy, priority assignmentpolicy, andwith threads facility, a process having a single thread correspondsto a process of a traditional operating system.
- Threads are oftenreferred to as lightweight processes and traditional processes arereferred to as heavyweight processes.

Motivations for Using Threads :

The main motivations for using a multithreaded processinstead of multiple single threaded processes for performing somecomputation activities are as follows:

1. The overheads involved in creating a new process are ingeneral considerably greater than those of creating a newthread within a process.
2. Switching between threads sharing the same address space isconsiderably cheaper than switching between processes thathave their own address space.
3. Threads allow parallelism to be combined with sequentialexecution and blocking system calls. Parallelism improvesperformance and blocking system calls make programmingeasier make programming easier.
4. Resource sharing can be achieved more efficiently and naturallybetween threads of a process than between processes becauseall threads of a process share the same address space.

These advantages are:

- The overheads involved in the creation of a new process andbuilding its execution environment are liable to be much greaterthan creating a new thread within an existing process.
- This ismainly because when a new process is created its address spacehas to be created from scratch, although a part of it might beinherited from the process's parent process.
- However, when a newthread is created, it uses the address space of its process that neednot be created from scratch.
- For instance, in case of a kernelsupported virtual memory system, a newly created process willincur page faults as date and instructions are referenced for the firsttime.

- Moreover, hardware caches will initially contain no datavalues for the new process, and cache entries for the process'sdata will be created as the process executes.
- These overheadsmay also occur in thread creation, but they are liable to be less.
- This is because when the newly created thread accesses code anddata that have recently been accessed by other threads within theprocess, it automatically takes advantage of any hardware or mainmemory caching that has taken place.
- Threads also minimize context switching time, allowing theCPU to switch from one unit of computation to another unit ofcomputation with minimal overhead. Due to the sharing of addressspace and other operating system resources among the threads ofa process, the overhead involved in CPU switching among peerthreads is very small as compared to CPU switching amongprocesses having their own address spaces.
- This is the reasonwhy threads are called lightweight processes.

True file service:

- It is concerned with the operation on individualfiles, such operations for accessing and modifying the data in filesand for creating and deleting.
- To perform these primitive fileoperations correctly and efficiently, typical design issues of a truefile service component include file accessing mechanism, filesharing semantics, file caching mechanism, file replicationmechanism, concurrency control mechanism, data consistency andmultiple copy update protocol, and access control mechanism.
- Note that the separation of the storage service from the true fileservice makes it easy to combine different methods of storage anddifferent storage media in a single file system.

Name service :

- IT provides a mapping between text names for filesand references to files, that is, file IDs. Text names are requiredbecause, file IDs are awkward and difficult for human users toremember and use. Most file systems use directories to performthis mapping.
- Therefore, the name service is also known as adirectory service.
- The directory service is responsible forperforming directory related activities such as creation and deletionof directories, adding a new file to a directory deleting a file from adirectory, changing the name of a file, moving a file from onedirectory to another, and so on.
- The design and implementation of the storage service of adistributed file system is similar to that of the storage service of acentralized file system.
- Readers interested in the details of thestorage service may refer to any good book on operating systems.
- Therefore, this chapter will mainly deal with the design andimplementation issues of the true file service component ofdistributed file systems.

PROCESS MIGRATION

Process migration is the relocation of a process from itscurrent location (the source node) to another node (the destinationnode). The flow of execution of a migrating process is illustrated in
Figure

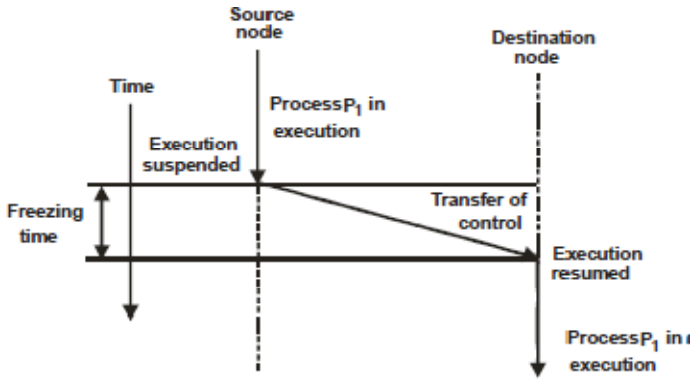


Fig : Flow of execution of a migrating process

- A process may be migrated either before it starts executingon its source node on during the course of its execution.
- Theformer is known as nonpreemptive process migration, and the latteris known as preemptive process migration.
- Preemptive processmigration is costlier than non-preemptive process migration sincethe process environment must also accompany the process to itsnew node for an already executing process.

Process migration involves the following major steps :
1. Selection of a process that should be migrated.
2. Selection of the destination node to which the selected processshould be migrated
3. Actual transfer of the selected process to the destination node.

- The first two steps are taken care of by the processmigration policy and the third step is taken care of by the processmigration mechanism.
- The policies for the selection of a sourcenode, a destination node, and the process to be migrated onresource management.

Threads :

The main motivations for using a multithreaded process instead of multiple single threaded processes for performing somecomputation activities are as follows:

1. The overheads involved in creating a new process are ingeneral considerably greater than those of creating a newthread within a process.
2. Switching between threads sharing the same address space isconsiderably cheaper than switching between processes thathave their own address space.Service makes it easy to combine different methods of storage anddifferent storage media in a single file system.

Name service :

- IT provides a mapping between text names for filesand references to files, that is, file IDs.
- Text names are requiredbecause, file IDs are awkward and difficult for human users toremember and use.
- Most file systems use directories to performthis mapping.
- Therefore, the name service is also known as adirectory service.
- The directory service is responsible forperforming directory related activities such as creation and deletionof directories, adding a new file to a directory deleting a file from adirectory, changing the name of a file, moving a file from onedirectory to another, and so on.
- The design and implementation of the storage service of adistributed file system is similar to that of the storage service of acentralized file system.
- Readers interested in the details of thestorage service may refer to any good book on operating systems.
- Therefore, this chapter will mainly deal with the design andimplementation issues of the true file service component of distributed file systems

SUBJECT TITLE: DISTRIBUTED COMPUTING

SUBJECT CODE : SCSX1028

UNIT V
DISTRIBUTED FILE SYSTEMS

DESIRABLE FEATURES OF A GOOD DISTRIBUTED FILE SYSTEM

A good distributed file system should have the features described below.

1. Transparency : The following four types of transparencies are desirable :

Structure transparency :

- Although not necessary, for performance, scalability and reliability reasons, a distributed file system normally uses multiple file servers.
- Each file server is normally a user process or sometimes a kernel process that is responsible for controlling a set of secondary storage device (used for file storage) of the node on which it runs.
- In multiple file servers, the multiplicity of file servers should be transparent to the clients of a distributed file system.
- In particular, clients should not know the number or locations of the file servers and the storage devices.
- Ideally, a distributed file system should look to its clients like a conventional file system offered by a centralized, time-sharing operating system.

FILE MODELS

- Different file systems use different conceptual models of afile.
- The two most commonly used criteria for file modeling arestructure and modifiability.

File models based on these criteria aredescribed below:

Unstructured and Structured Files :

- According to the simplest model, a file is an unstructuredsequence of data. In this model, there is no substructure known tothe of each file of the file system appears to the file server as anuninterrupted sequence of bytes.
- The operating system is notinterested in the information stored in the files, the interpretation ofthe meaning and structure of the data stored in the files are entirelyup to the application programs.
- UNIX and MS-DOS use this fiemodel.
- Another file model that is rarely used nowadays is thestructured file model. In this model, a file appears to the file serveras an ordered sequence of records.
- Records of different files of thesame file system can be of different size.
- Therefore, many types offiles exist in a file system, each having different properties.

- In thismodel a record is the smallest unit of file data that can beaccessed, and the files system read or write operations are carriedout on a set of records.
- Structured files are again of two types – files with nonindexedrecords and files with indexed records.
- In the formermodel, a file record is accessed by specifying its position within thefile, for example, the fifth record from the beginning of the file or thesecond record from the end of the file. In the latter model, recordshave one or more key fields and can be addressed by specifyingthe values of the key fields.
- In file systems that allow indexedrecords, a file is maintained as a B-tree or other suitable datastructure or a hash able is used to locate records quickly.
- Most modern operating systems use the unstructured filemodel.
- This is mainly because sharing of a file by differentapplications is easier with the unstructured file model as comparedto the structured file model.
- Since a file has no structure in theunstructured model, different applications can interpret the contentsof a file m different ways.
- In addition to data items, files also normally have attributes.
- A file's attributes are information describing that file.
- Each attributehas a name and a value.
- For example, typical attributes of a filemay contain information such as owner, size, access permissions,date of creation, date of last modification, and date of last access.
- Users can read and update some of the attribute values using theprimitives provided by the file system.
- Notice, however, thatalthough a user may update the value of any attribute, not allattributes are user modifiable.
- For example, a user may update thevalue of the access permission attribute, but he or she cannotchange the value of the size or date of creation attributes.
- The types of attributes that can be associated with a file are normallyfixed by the file system.
- However, a file system may be designedto provide the flexibility to create and manipulate user definedattributes in addition to those supported by the file system.
- File attributes are normally maintained and used by thedirectory service because they are subject to different accesscontrols than the file they describe.
- Notice that although fileattributes are maintained and used by the directory service, theyare store with the corresponding file rather than with the file namein the directory.
- This is mainly because many directory systemsallow files to be referenced by more than one name.

Mutable and Immutable Files :

- According to the modifiability criteria, files are of two types –mutable and immutable.
- Most existing operating systems use themutable file model.
- In this model, an update performed on a fileoverwrites on its old contents to produce the new contents.
- That is,a file is represented as a single stored sequence that is altered byeach update operation.

- On the other hand, some more recent file systems, such asthe Cedar File System (CFS), use the immutable file model.
- In thismodel, a file cannot be modified once it has been created except tobe deleted.
- The file versioning approach is normally used toimplement file updates, and each file is represented by a history ofimmutable versions.
- That is, rather than updating the same file, anew version of the file is created each time a change is made to thefile contents and the old version is retained unchanged.
- In practice,the use of storage space may be reduced by keeping only a recordof the differences between the old and new versions rather thancreating the entire file once again.
- Gifford et al. emphasized that sharing only immutable filesmakes it easy to support consistent sharing.
- Due to this feature, itis much easier to support file caching and replication in adistributed system with the immutable file model because iteliminates all the problems associated with keeping multiple copiesof a file consistent.
- However, due to the need to keep multipleversions of a file, the immutable file mode, suffers from twopotential problems – increased use of disk space and increaseddisk allocation activity.
- Some mechanism is normally used toprevent the desk space from filling instantaneously.

FILE ACCESSING MODELS

- The manner in which a client's request to access a file isserviced depends on the file accessing model used by the filesystem.
- The file accessing model of a distributed file systemmainly depends on two factors – the method used for accessingremote files and the unit of data access.

Byte level transfer model :

- In this model, file data transfers acrossthe network between a client and a server take place in units ofbytes.
- This model provides maximum flexibility because it allowsstorage and retrieval of an arbitrary sequential subrange of a file,specified by an offset within a file, and a length.
- The maindrawback of this model is the difficulty in cache management due tothe variable length data for different access requests.
- TheCambridge File Server [Dion 1980, Mitchell and Dion 1982,Needham and Herbert 1982] uses this model.

Record level transfer mode :

- The three file data transfer modelsdescribed above are commonly used with unstructured file models.
- The record level transfer model is suitable for use with those filemodels in which file contents are structured in the form of records.
- In this model, file data transfers across the network between aclient and a server take place in units of records.

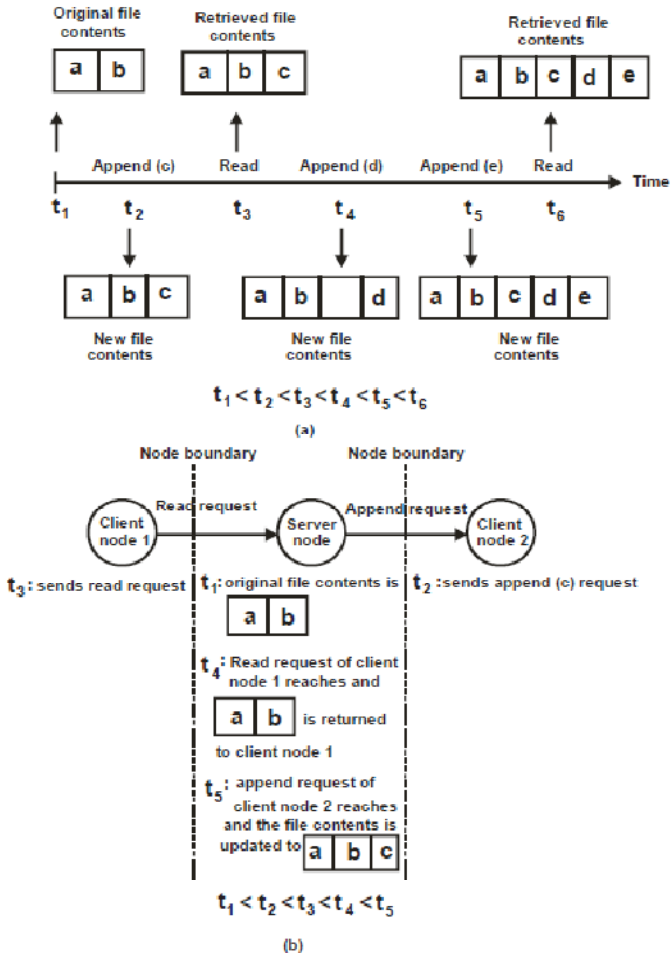
- The ResearchStorage System (RSS) [Gray 1978 Gray et al. 1981], whichsupports complex access methods to structured and indexed files,uses the record level transfer mode.

FILE SHARING SEMANTICS

- A shared file may be simultaneously accessed by multipleusers.
- In such a situation, an important design issue for any filesystem is to clearly defined when modifications of file data made bya user are observable by other users.
- This is defined by the type offile sharing semantics adopted by a file system

1. UNIX semantics :

- This semantics enforces an absolute timeordering on all operations and ensures that every read operation ona file sees the effects of all previous write operations performed onthat file **[Fig.]**.
- n particular, writes to an open file by a userimmediately become visible to other users who have this file openat the same time.



(a) Example of UNIX file sharing semantics (b) an example explaining why it is difficult to achieve UNIXsemantics in a distributed file system even what the shared fileis handled by a single server

- The UNIX semantics is commonly implemented in filesystems for single processor systems because it is the mostdesirable semantics and also because it is easy to serialize all read/write requests.
- However, implementing UNIX semantics ina distributed file system is not an easy task.
- One may think thatthis semantics can be achieved in a distributed system bydisallowing files to be cached at client nodes and allowing a sharedfile to be manage by only one file server that processes all read andwrite requests for the file strictly in the order in which it receivesthem.
- However, even with this approach, there is a possibility that,due to network delays, client requests from different nodes mayarrive and get processed at the server node in an order differentfrom the actual order in which the requests were made.
- Furthermore, having all fie access requests processed by a singleserver and disallowing caching on a client nodes is not desirable inpractice due to poor performance, poor scalability, and poorreliability of the distributed file system.
- Therefore, distributed filesystems normally implement a more relaxed semantics of filesharing.
- Applications that need to guarantee UNIX semantics forcorrect functioning should use special means (e.g. locks) for thispurpose and should not rely on the underlying semantics of sharingprovided by the file system.

FILE CACHING SCHEME

- File caching has been implemented in several file systemsor centralized time sharing systems to improve file I/Operformance.
- The idea in file caching in these systems is to retainrecently accessed file data in main memory, so that repeatedaccesses to the same information can be handled withoutadditional disk transfers.
- Because of locality in file access patterns,file caching reduces disk transfers substantially, resulting in betteroverall performance of the file systems.
- The property of locality infile access patterns can as well be exploited in distributed systemsby designing a suitable file caching scheme.
- In addition to betterperformance, a file caching scheme for a distributed file systemmay also contribute to its scalability and reliability because it ispossible to cache remotely located data on a client node.
- Therefore, every distributed file system in serious use today usessome form of file caching.
- Even AT & T's Remote File System(RFS) which initially avoided caching to emulate UNIX semantics,now uses it.
- In implementing a file caching scheme for a centralized filesystem one has to make several key decisions, such as thegranularity of cached data (large versus small), cache size (largeversus small, fixed versus dynamically changes, and

thereplacement policy. A good summary of these design issues ispresented in [Smith 1+82].

- In addition to these issues, a filecaching scheme for a distributed file system should also addressthe following key decisions:
    1. Cache location
    2. Modification propagation
    3. Cache validation

These three design issues are described below.

Cache Location :

- Cache location refers to the place where the cached data isstored.
- Assuming that the original location of a file is on its server'sdisk, there are three possible cache locations in a distributed filesystem.
- In this approach, a read quorum of *r* votes is collected toread a file and a write quorum of *w* votes to write a file.
- Since thevotes assigned to each copy are not the same, the size of a read /write quorum depends on the copies selected for the quorum.
- Thenumber of copies in the quorum will be less if the number of votesassigned to the selected copies is relatively more.
- On the otherhand, the number of copies in the quorum will be more if thenumber of votes assigned to the selected copies is relatively less.
- Therefore, to guarantee that there is a non-null intersectionbetween every read quorum and every write quorum, the values of*r* and w are chosen such that *r* + *w* is greater than the total numberof votes (*v*) as to the file (*r* +*w* +*v*). Here, *v* is the sum of thevotes of all the copies of the file.

Modification Propagation:

- In the file system in which the cache is located on clients'node; a file's data may simultaneously be cached on multiplenodes.
- In such a situation, when cache of all these nodes containsexactly the same copy of the file data, we say that the caches areconsistent.
- It is possible for the cache to become inconsistentprovided the file data is modified by one of the clients' and thecorresponding data cached at the other nodes are not changed ordiscarded.
- Keeping file data cached at multiple client nodes consistentis an important design issue in those distributed file systems thatuse client caching.
- A variety of approaches handle this issue havebeen proposed and implemented.
- These approaches depend onthe schemes used for the following cache design for distributed filesystem.
  1) When to propagate modifications made to a cached data tocorresponding file server.
  2) How to verify the validity of cached data.

Cache Validation Scheme:

- A file data may simultaneously reside in the cache ofmultiple nodes.

- The modification propagation policy only specifieswhen the master copy of a file at a server node is updated uponmodification of a cache entry.

- It does not tell anything about when the file data residing inthe cache of other nodes was updated.
- As soon as other nodes get updated, the client's databecome outdated or stale.
- Thus the consistency of the clients'cache has to be checked and must be consistent with the mastercopy of the data.

Validation is done in two ways:

1) **Client initiated approach:**
   - Here client checks for new updates before it accesses itsdata or it goes with the periodic checking mechanism i.e.client checks for updates after regular intervals of time.
   - Herethe pull mechanism is implemented where the client Pulls forupdates.

2) **Server initiated approach:**
   - Here the server is responsible for sending periodic updatesto all its clients.
   - The Push protocol is user where she serverpushes the new updates to all its clients.

FAULT TOLERANCE

- Fault tolerance is an important issue in the design of adistributed file system.
- Various types of faults could harm theintegrity of the data stored by such a system.
- For instance, aprocessor loses the coments of its main memory in the event of acrash.
- Such a failure could result in logically complete butphysically incomplete file operations, making the data that arestored by the file system inconsistent.
- Similarly, during a requestprocessing, the server or client machine may crash, resulting in theloss of state information of the file being accessed.
- This may havean uncertain effect on the integrity of file data.
- Also, other adverseenvironmental phenomena such as transient faults (caused byelectromagnetic fluctuations) or decay of disk storage devices mayresult in the loss or corruption of data stored by a file system.
- Aportion of a disk storage device is said to be 'decay'. The data onthat portion of the device are irretrievable.
- The primary file properties that directly influence ability of adistributed file system to tolerate faults are as follows.

1. Availability :
- Availability of a file refers to the fraction of time forwhich the file is available for use.
- Note that the availabilityproperty depends on the location of the file and the locations ofits clients (users).

- For example, if a network is partitioned dueto a communication link failure, a file may be available to theclients of some nodes, but at the same time, it may not beavailable to the clients of other nodes.
- Replication is a primarymechanism for improving the availability of a file.

2. Robustness :

- Robustness of a file refers to its power to survivecrashes of the storage device and decays of the storagemedium on which it is stored. Storage devices that are implemented by using redundancy techniques, such as stablestorage device, are often used to store robust files.
- Note that arobust file may not be available until the faulty component hasbeen recovered.
- Furthermore, unlike availability, robustness isindependent of either the location of the file or the location of itsclients.
- On the other hand, if a failure occurs that causes asubtransaction to abort before its completion, all of its tentativeupdates are undone, and its parent is notified.
- The parent maythen choose to continue processing and try to complete its taskusing an alternative method or it may abort itself.
- Therefore, theabort of a subtransaction may not necessarily cause its ancestorsto abort.
- However, if a failure causes an ancestor transaction toabort, the updates of all its descendant transactions (That havealready committed) have to be undone.
- Thus no updatesperformed within an entire transaction family are made permanentuntil the top level transaction commits.
- Only after the top leveltransaction commits is success reported to the client.

Advantages of Nested Transactions :

- Nested transactions facility is considered to be an importantextension to the traditional transaction facility (especially indistributed system) due to its following main advantages:

1. It allows concurrency within a transaction. That is a transactionmay generate several subtransactions that run in parallel ondifferent processors. Notice that all children of a parenttransaction are synchronized so that the parent transaction stillexhibits serializability.

2. It provides greater protection against failures, in that it allowscheckpoints to be established within a transaction. This isbecause the subtransactions of a parent transaction failindependently of the parent transaction and of one teacher.Therefore, when a subtransaction aborts, its parent can stillcontinue and may fork alternative subtransaction in place of thefailed subtransaction in order to complete its task.

ATOMIC TRANSACTIONS :

- An atomic transaction (or justtransaction for shore) is a computation consisting of a collectionof operation that take place indivisibly in the presence of failuresand concurrent computations.
- That is, either all of theoperations are performed successfully or none of their effectsprevails, other processes executing concurrently cannot modifyor observe intermediate states of the computation.
- Transactions help to preserve the consistency of a set of shareddate objects (e.g. files) in the face of failures and concurrentaccess.
- They make crash recovery much easier, becausetransactions can only end in two states :
- Either all theoperations of the transaction are performed or none of theoperations of the transaction is performed.
- Before a transaction iscompleted, the system subsequently restores any data objectsthat were undergoing modification to their original states.
- Noticethat if a system does not support a transaction mechanism,unexpected failure of a process during the processing of anoperation may leave the data objects that were undergoingmodification in an inconsistent state.
- Therefore, withouttransaction facility, it may be difficult or even impossible in somecases to roll back (recover) the data objects from their currentinconsistent states to their original states.