ICS 2208: OPERATING SYSTEMS II

BIT2110/ICS 2208: OPERATING SYSTEMS II

Course Outline:

Principles of Distributed Operating Systems. Networking Protocols. Distributed file systems. Remote Procedure Calls, Distributed process management. Load balancing and process migration. Programming languages for distributed systems. Client/server distributed model. Graphical user interfaces. Systems Comparisons. Performance Evaluation. Case Studies: Current Standards and Novel Systems Under Development.

Core text: Distributed Operating systems by Pradeep K. Sinha, 5th ed, Any other DOS text.

Week	Topic	Subtopics
1 – 2	**Introduction**	➢ **What is distributed computing systems (DCS)** ➢ **Evolution of DCS and DCS models** ➢ **Why DCS are gaining popularity** ➢ **What is a DOS and the design issues** ➢ **Summary**
3	**Networking**	➢ **Introduction** ➢ **Communication protocols** ➢ **Summary** ➢ **CAT1 + Assignment1**
4 – 5	**Message passing**	➢ **Introduction** ➢ **Desirable features of a good message-passing system and issues in IPC by message-passing** ➢ **Synchronization, buffering and multidatagram messages** ➢ **Encoding and decoding of message data** ➢ **Failure handling** ➢ **Group communication** ➢ **Summary**
6 – 7	**Remote procedure calls**	➢ **Introduction** ➢ **The RPC models** ➢ **Transparency of RPC and implementing RPC mechanism** ➢ **Stub generation, RPC messages** ➢ **Server management** ➢ **Call semantics and communication protocols for RPC etc** ➢ **Summary** ➢ **CAT2 + Assignment2**
8	**Resource management**	➢ **Introduction** ➢ **Desirable features** ➢ **Methodologies for resource management** ➢ **Summary**
9	**Process management**	➢ **Introduction** ➢ **Process migration** ➢ **Threads** ➢ **Summary**
10 – 11	**Distributed file systems**	➢ **Introduction** ➢ **Desirable features** ➢ **File models and file access models** ➢ **File-sharing semantics** ➢ **File-caching schemes** ➢ **File replication, fault tolerance etc** ➢ **Summary** ➢ **CAT3 + Assignment3**
12,13 &14	**Case studies and revision**	➢ **Issues of security, naming and sample studies**
15/16	**Exams**	➢ **Exams**

CHAPTER 1: INTRODUCTION

The advancement in microelectronics technology resulting in fast, inexpensive processors, and advancements in communication resulting in cost effective and highly efficient computer networks have had a net result in changing the price performance ratio to favour the use of interconnected multiple processors in place of a single, high speed processor.

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

1. **Tightly coupled systems (see Fig. 1.1)** – have a single system wide primary memory shared by all the processors.
2. **Loosely coupled system (see fig. 1.2)** – each processor has its own local memory.

See diagrams below.

Fig. 1.1

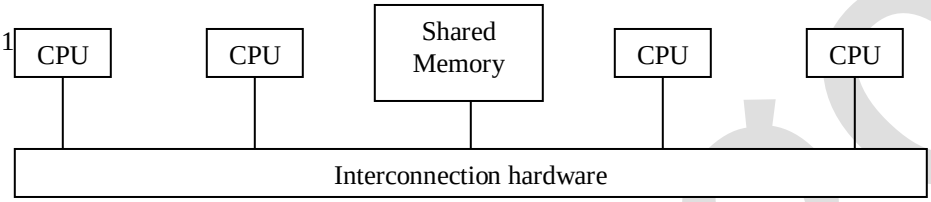
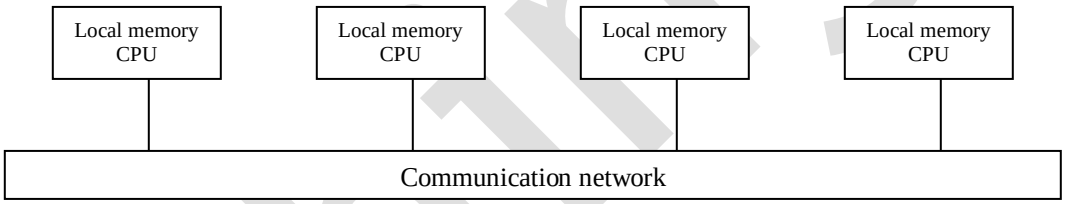


Fig. 1.2



The tightly coupled systems are known as *parallel processing systems*.

NB:

- Processors of distributed systems can be located far from each other to cover a wider geographical area (unlike in parallel systems).
- In tightly coupled systems, the number of processors that can be usefully deployed is small and limited by the bandwidth of the shared memory. (Distributed systems are almost unlimited).

Definition

A Distributed Computing System (DCS) is a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals and communication between any two processors of the system takes place by message passing over a communication network (i.e. for a particular processor, its own resources are local).

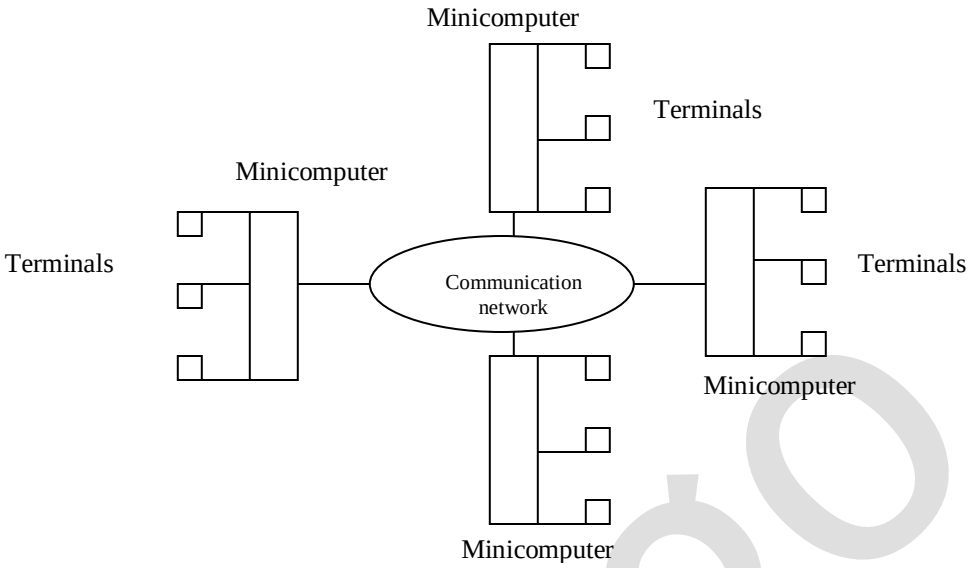
Distributed computing system models:

Used for building distributed computing systems and can be broadly classified into five as below: -

i. **Minicomputer model:**

- Is a simple extension of the centralized time-sharing model
- Consists of a few mini computers (or super computers) interconnected by a communication network.
- Each minicomputer has multiple users simultaneously logged on.
- Used when resource sharing (e.g., databases of different types, each located on different machine) with remote users is desired.
- ARPAnet is an example.

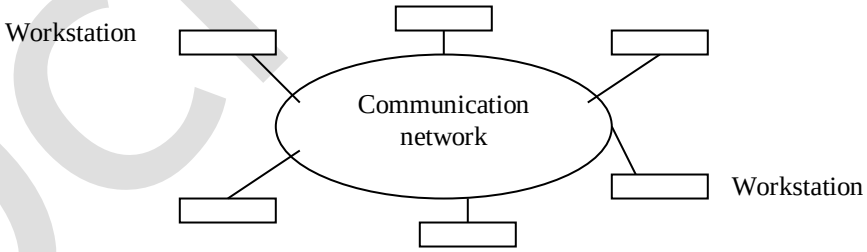
See Fig. 1.3 below.



ii. **Workstation Model**

- Made of several work stations interconnected by a communication network.
- Each workstation is equipped with its own disk and serves a single user.
- Has the disadvantage that some stations lie idle (especially at night) thus wasting CPU as a resource.
- Solution is to interconnect the workstations by a high-speed LAN so that they may have sufficient processing power i.e., jobs partly transferred to another station for execution and then results returned to the original station.
- The model however is not easy to implement (give as a task) e.g., how do you find an idle workstation? what happens if the workstation initiates its own work? etc
- Sprite system is an example.

Fig. 1.4

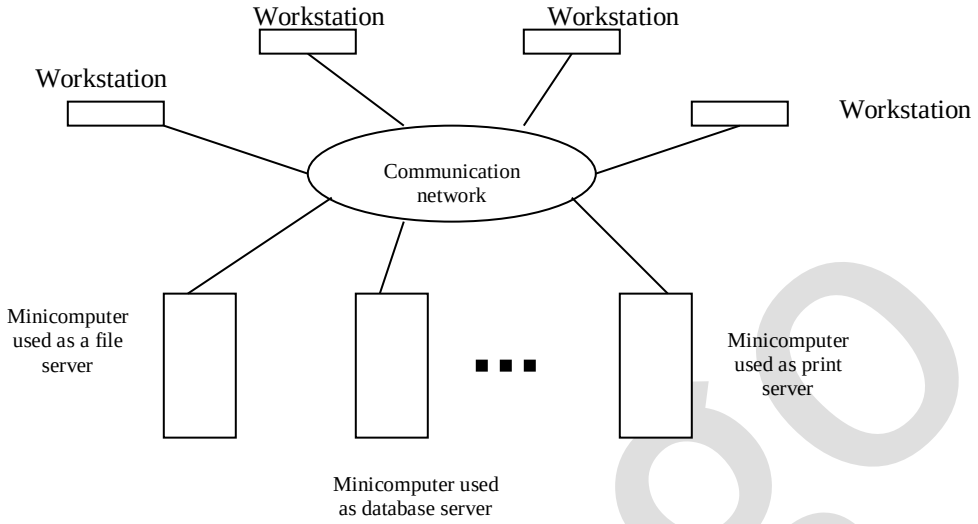


iii. **Workstation – server model:**

- Is a network of personal workstations each with its own disk and a local file system.
- Consists of a few minicomputers and several workstations (some diskless and some diskfull) interconnected by a communication network.
- The minicomputers are basically used to provide one or more types of services e.g., file system, database service and print service.
- Compared to the workstation model, it has the following advantages:
    a. 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 diskfull workstations with each workstation having a small slow disk.
    b. Diskless are preferred to diskfull workstations from a systems maintenance point of view e.g., in terms of h/ware, backup, s/ware installation etc.
    c. 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 w/station a user is currently logged onto.
    d. Does not need a process migration facility

e. User has guaranteed response time since w/stations are not used for executing remote processes (but the model does not use the processing capability of idle workstations).
f. V-system is an example.

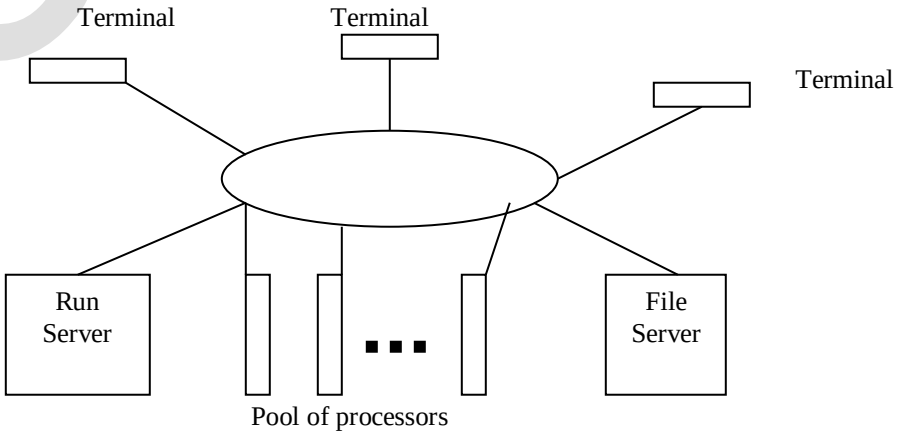
Fig. 1.5: Workstation-Server Model



iv. **Processor-Pool Model:**

- Based on the observation that most of the time a user does not need any computing power but once in a while may need a very large amount of the power for a short time e.g., when recompiling a program.
- The processors are thus pooled together to be shared by the users as needed.
- The pool of processors is a large number of micro and minicomputers, with each processor having its own memory to load and run a system program or an application program.
- Processors have no directly attached terminals i.e.; terminals are attached to the network via special devices.
- They may be small diskless workstations or graphic terminals.
- The run server (special server) manages and allocates the processors in the pool to different users on a demand basis.
- No user has a home machine i.e., user logs to system as a whole.
- Compared to workstation-server model, it allows better utilization of the available processing power.
- Flexible in that system's services can be easily expanded without the need to install any more computers i.e., the processors in the pool can be allocated to act as extra servers or to provide new services.
- Unsuitable for high-performance interactive applications due to slow speed of communication between the computer and the terminal.
- Amoeba, Cambridge DCS are examples

Fig. 1.6



v. **Hybrid-Model:**
- Combines the features of both the w/station-server model and the processor-pool model. The processors can then be allocated dynamically for work too large for w/stations and give guaranteed response to interactive jobs by allowing them to be processed on local w/stations of user, but is much more expensive than either of the two.

Why Distributed Systems Are Becoming Popular

1. **Inherently Distributed Applications:** Many applications are by nature, inherently distributed thus requiring distributed computing for their realization e.g., for collecting, preprocessing and accessing data. Examples include computerized world-wide airline reservation system, banking system, loaning system etc.
2. **Information Sharing Among Distributed Users:** there is desire for efficient person-to-person communication facility by sharing information over great distances e.g., two far-off users can work on the same project.
3. **Resource Sharing:** resources both h/w and s/w can be shared.
4. **Better Price-Performance Ratio:** they have this quality as compared to the centralized systems due to the rapidly increasing power and reduction in the price of microprocessors together with the increasing speed of communication networks. They facilitate resource sharing among multiple computers.
5. **Shorter response time and higher throughput:** They have better performance due to multiple processors as compared to single-processor centralized systems.
6. **Higher reliability:** due to the multiplicity of processors and storage devices, multiple copies of critical information is maintained and redundancy achieved. Also, geo graphical distribution limits scope of failures caused by natural disasters. Its important aspect is "availability" i.e., the fraction of time a system is available for use.
7. **Extensibility and Incremental growth:** they are capable of incremental growth i.e., additional resources both s/w and h/w can be added. Distributed systems with these qualities are referred to as "open distributed systems."
8. **Better flexibility in meeting users' needs:** a distributed system may have a pool of different types of computers, so that the most appropriate one can be selected for processing a user's job depending on the nature of the job.

Q: What is A Distributed Operating System (DOS)?
Definition: *"Is one that looks to its users like an ordinary centralized OS but runs on multiple, independent CPUs." (Tanenbaum and Van Renesse [1985])*

The OS commonly used for DCS are broadly classified into two namely "network OS" and "distributed OS". Three features that differentiate them are:

a. **System image** – in a network OS, users view the DCS as a collection of distinct machines connected by a communication subsystem i.e., users are aware that multiple computers are being used, but DOS hides the existence of multiple computers and provides a single-system image to its users as a "virtual uniprocessor."
b. **Autonomy** – for NOS, each computer has its own local OS and functions independent of others except when they must intercommunicate whereby they must use a mutually agreed on communication protocol. For a DOS, there is a single system-wide OS and each computer runs a part of this global OS. Processes and resources are managed globally and there is a single set of system calls (globally valid) available on all computers of the DCS.
c. **Fault tolerance capability** – this is usually high for a DOS than a NOS e.g., a 10% loss in NOS affects 10% users but in DOS only 10% loss in performance is experienced,

Issues in Designing a DOS
A DOS is more difficult to design than a centralized OS (COS) for several reasons e.g., in COS, it is assumed that the OS has access to complete and accurate information about the environment in which it is functioning but DOS must be designed with the assumption that complete information will never be available i.e., resources are physically separated, there are no common clocks among the multiple processors, delivery of messages is delayed and messages could even be lost.

The following therefore are some of the key design issues:

1. **Transparency:** One main aim of DOS is to display the existence of multiple computers as invisible (transparent) and provide a single system image to users (virtual uniprocessor). The eight forms of transparency include:

a. **Access transparency –** users should not need or be able to recognize whether a resource, both h/w and s/w is remote or local i.e., the user interface should not distinguish between remote and local resources.
b. **Location transparency –** which has two aspects, *name transparency*, i.e., the name of a resource s/w and h/w should not reveal any hint as to the physical location of the resource i.e., name should be independent of the physical connectivity/topology or the current location of the resource, and must be unique system wide. *User mobility*, i.e., no matter which machine a user is logged onto, he/she should be able to access a resource with the same name.
c. **Replication transparency –** All DOS have the provision to create replicas of files and other resources on different nodes and both the existence of multiple copies of a replicated resource and the replication activity should be transparent to users.
d. **Failure transparency –** this should mask from the users, partial failures of the system, e.g., a communication link failure, a machine failure or a storage device crash. The DOS should continue to function during partial failures, probably in a degraded form. Examples are fail-stop and byzantine failures.
e. **Migration transparency –** aims at ensuring that the movement of an object (e.g., process or file) is handled automatically by the system in a user-transparent manner. To achieve this objective, the following three issues must be looked into:
    i. Migration decisions as to which object is to be moved should be made automatically by the system
    ii. Migration of an object should not require any change in its name
    iii. When the migrating object is a process, the interprocess communication mechanisms should ensure that a message sent to the migrating process reaches it without the need for the sender process to resend it if the receiver process moves to another node before the message is received.
f. **Concurrency transparency –** allows each user to feel that he/she is the sole user to the system. To achieve this, the resource sharing mechanisms of the DOS must display the following four properties:
    i. An event-ordering property ensures that all access requests to various system resources are properly ordered to provide a consistent view to all users of the system.
    ii. A mutual-exclusion property ensures that at any time at most one process accesses a shared resource, which must not be used simultaneously by multiple processes if program operation is to be correct.
    iii. A non-starvation property ensures that if every process that is granted a resource, which must not be used simultaneously by multiple processes, eventually releases it, every request for that resource is eventually granted.
    iv. A non-deadlock property ensures that a situation will never occur in which competing processes prevent their mutual progress even though no single one requests more resources than available in the system.
g. **Performance transparency** – allows the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system e.g., processing must be evenly distributed with jobs.
h. **Scaling transparency** – allows the system to expand in scale without disrupting the activities of the users

2. **Reliability:** is possible in DOS due to the existence of multiple instances of resources. A fault may occur (either mechanical or algorithmic) generating an error. System failures could either be *fail-stop* where system stops functioning after changing to a state in which a failure can be detected or *Byzantine failure* where system continues to function but produces wrong results.

   To achieve higher reliability, the fault handling mechanism must be designed to avoid faults, tolerate faults and detect and recover from faults as explained below.
   a. **Fault avoidance** – deals with designing the components of the system such that the occurrence of faults is minimized
   b. **Fault tolerance** – is the ability of a system to continue functioning in the event of partial failure. This can be achieved through
       i. **Redundancy techniques** – this avoids single points of failure by replicating critical h/w and s/w components, such that if one fails, the others can be used to continue.
       ii. **Distributed control** – many of the particular algorithms or protocols used in a DOS must employ a distributed control mechanism to avoid single points of failure.

c. **Fault detection and recovery** – is the use of h/w and s/w mechanisms to determine the occurrence of a failure and then to correct the system to a state acceptable for continued operation. Techniques include:
    i. **Atomic transactions** – is a computation consisting of a collection of operations that take place indivisibly in the presence of failures and concurrent computations.
    ii. **Stateless servers** – the stateless service paradigm makes crash recovery very easy because no client state information is maintained by the server.
    iii. **Acknowledgements and time-out-based retransmissions of messages** – the receiver must return an acknowledgement message for every message received, and if sender does not receive any acknowledgement for a message within a fixed timeout period, it assumes that the message was lost and retransmits.

3. **Flexibility:** allows the following to be achieved:
    i. **Ease of modification** – incorporating changes in a user-transparent manner or with minimum interruption on users.
    ii. **Ease of enhancements** – e.g., adding new functionalities from time to time to make it more powerful and easy to use.

4. **Performance:** its performance must be at least, as good as that for a centralized system. Design principles considered include:
    i. **Batch if possible** – e.g., transferring large chunks of data at once instead of individual pages.
    ii. **Cache whenever possible** – makes data available whenever it is currently being used thereby saving time and bandwidth. Reduces contention on centralized systems.
    iii. **Minimize copying of data** – e.g., moving data in and out of buffers.
    iv. **Minimize network traffic** – e.g., migrating a process closer to the resources it is using most heavily.
    v. **Take advantage of fine-grain parallelism for multiprocessing** –e.g., using threads for server processes thereby allowing the servers to simultaneously service requests from several clients.

5. **Scalability**: is the capability of a system to adapt to increased service load. Guiding principles for item design include:
    a. **Avoid centralized entities: -** use of centralized entities e.g., a single central file server or database for the entire system makes the DS non-scalable due to:
        i. Failure of centralized entity brings down whole system
        ii. Performance becomes a system bottleneck when contention for it increases with increasing number of users.
        iii. Capacity of the network that connects the centralized entity with other nodes of the system often gets saturated when the contention for the entity increases beyond a certain level.
        iv. In a WAN comprising several LANS, it is inefficient to always get a particular type of request serviced at a server node that is several gateways away.
    b. **Avoid central algorithms**: centralized algorithms operate by collecting information from all nodes, processing it on a single node and then distributes the result to other nodes. It suffers similar problems mentioned above.
    c. **Perform most operations on client workstations: -**Perform client operations on the work station rather than on a server machine. Server is common resource and so its cycles are more precious. It allows graceful degradation of the system performance as it grows in size by reducing contention for shared resources.

6. **Heterogeneity**: - A heterogeneous DS consists of interconnected sets of dissimilar hardware or software system. It allows users flexibility to use various platforms for their work.

7. **Security**: - Enforcing it is difficult due to lack of single point of control and the use of insecure networks for data communication. Since the client–server model is often used for requesting and providing services, when a client sends a request message to a server, the server must have some

way of knowing whom the client is, which is not that simple. To enforce security hence requires the following additional requirements, as compared to a centralized system: -

i. it should be possible for the sender of the message to know that the message was received by the intended receiver.
ii. It should be possible for the receiver to know that the message was sent by the genuine sender.
iii. It should be possible for both the sender and receiver to be guaranteed that the contents of the message were not changed while it was in transit - integrity.

8. **Emulation of existing operating systems**: - For commercial success, its vital that a newly designed DOS be able to emulate existing popular OS e.g., Unix so that, new software can be written using the system call interface of the new OS to take full advantage of its special features of distribution, while old software can also be run on without rewriting them.

CHAPTER 2: NETWORK PROTOCOLS

The term protocol refers to rules and conventions i.e., agreements needed for communication between the communicating parties e.g., agreements are needed for:

- Handling duplicate messages
- Avoiding buffer overflows
- Assuring proper message sequencing/sessioning etc.

Computer networking is implemented using the concept of "layered protocols" and the main reasons for layering include:

i. Protocols are fairly complex thus layering makes implementation more manageable
ii. Layering provides well defined interfaces between the layers so that a change in one layer does not affect an adjacent layer
iii. Layering allows interaction between functionally pared layers in different locations.

Protocols for network systems

Network systems communication protocols allow remote computers to communicate with each other and allow users to access remote resources, but those for distributed systems enhance it by bringing in the concept of transparency.

Example Network Protocols

The OSI/ISO Reference Model

The ISO developed a reference model, the OSI. It is simply a guide and not a specification that provides framework in which standards can be developed for the services and protocols at each layer.

The seven-layered structure is discussed below:

7. Application layer
6. Presentation layer
5. Session layer
4. Transport layer
3. Network layer
2. Data Link layer
1. Physical layer

1. **Physical layer:** responsible for transmitting raw bit streams between two sites i.e., it deals with the mechanical, electrical, procedural and functional characteristics of transmissions of raw bit streams between two sites. RS232-c is a popular standard for serial communication lines at this layer.

2. **Data Link layer:** detects and corrects any errors in the transmitted data. Partitions data into frames to allow for detection/correction of errors on independent frames. It also performs flow control.

3. **Network layer:** sets up a logical path between two sites for communication to take place. It encapsulates frames into packets for transmission from one site to another using a high-level addressing and routing scheme i.e., routing is the primary job of this layer. Two protocol examples include X.25 (connection-oriented) and IP (connectionless).

4. **Transport layer:** provides site-to-site communication and hides all the details of the communication subnet from the session layer by providing a network-independent transport service i.e., it accepts messages of arbitrary lengths from session layer, segments them into packets, submits them to network layer and finally reassembles the packets at the destination. TCP and User Datagram Protocol (UDP) are two popular transport layer protocols.

5. **Session layer:** provides the means by which presentation entities can organize and synchronize their dialog and manage their data exchange i.e.
   - Allows two parties to authenticate each other before establishing a dialogue
   - Specifies dialogue type e.g., one-way, two-way alternate or two-way simultaneous etc.

6. **Presentation layer:** represents message information to communicating application layer entities in a way that preserves meaning while resolves syntax differences.

7. **Application layer:** provides services that directly support the end users of the network i.e., basically a collection of miscellaneous protocols for various commonly used applications e.g., email, file transfer, remote login etc.

NB: layer 1-3 are in hardware, 4-5 in operating system 6 in library subroutines and 7 in user program.

Case study:
The Internet Protocol suite:

5. Application layer	FTP, TELNET
4. Transport layer	TCP, UDP
3. Network layer	IP, ICMP
2. Data Link layer	ARP, RARP
1. Physical layer	ETHERNET, TOKEN RING ETC

Of the available protocol suites for networks, e.g., IP, XNS, SNA etc, IP has emerged as most popular and widely used due to:

- Suitability for both LANs and WANs
- Can be implemented on all types of computers
- Not vendor specific

It has five layers as shown in the above diagram.

TASK: explore this as a task.

Protocols for Distributed Systems
Network protocols discussed above offer adequate support for traditional network applications e.g., file transfer, remote login, but not suitable for distributed systems/applications. Special requirements for DS compared to network systems include:

- **Transparency -** communication protocols for DS must use location-independent process identifiers that do not change even when a process migrates from one node to another (i.e., support process migration)

- Client-server based communication: must have simple connectionless protocol having features to support request/response behaviour.

- **Group communication -** made more flexible.

- **Security**: enhanced.

- **Network management -** should be automatic to respond to dynamic changes in network traffic.

- **Scalability -** the communication protocol must scale well and be efficient for both LAN and WAN i.e., similar semantics.

CHAPTER 3: MESSAGE PASSING

Introduction:
Each computer of a distributed system may have a *resource manager* process to monitor the current status of usage of its local resources and the resource managers of all the computers may communicate with each other in order to dynamically balance load among the computers. Thus, a DOS needs to provide inter process communication (IPC) mechanism to enable this. Two main methods of information sharing are:

i. Original sharing (shared-data approach).
ii. Copy sharing (message-passing approach).

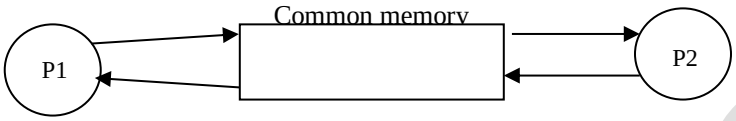


Fig. 2 (a)

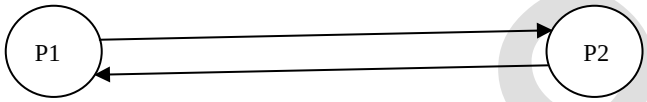


Fig. 2 (b)

Mechanism *a* places the shared data via common memory area accessible to all processes involved in IPC while *b* provides for the information to be physically copied from the sender to the receiver, by transmitting the data to be copied in the form of *message*. Mechanism *b* is used in distributed systems (DS) since computers in a network do not share memory.

Definition:
A "message-passing subsystem" is one of a DOS that provides a set of message-based IPC protocols and does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers. (Allows programmers to use simple primitives e.g., "send" and "receive" to communicate).

Desirable Features of a Good Message-Passing System

i. **Simplicity -** Should be simple and easy to use, i.e., constructed new applications should communicate with the existing ones using the existing primitives and programmer should be able to designate the various modules in the DS to allow them to intercommunicate without problems.
ii. **Uniform semantics -** a message-passing mechanism in a DS may take the forms
    a. Local communication – same node
    b. Remote communication – different nodes

Thus, semantics for both mechanisms should be as close as possible to ensure the system is easy to use.

iii. **Efficiency -** may take the form of reducing the number of message exchanges as far as practicable via
    a. Avoiding the cost of establishing and terminating connections between the same pair of processes for each and every massage exchange between them.
    b. Minimizing the costs of maintaining the connections.
    c. Piggybacking (take credit, share credit, and share the accomplishments) of acknowledgement of previous messages with the next message during a connection between sender and receiver that involves several message exchanges.
iv. **Reliability -** the system should cope with e.g., failures and guarantee message delivery through acknowledgments and retransmissions. Also eliminating duplicate messages via assigning appropriate sequence numbers.
v. **Correctness -** especially important for a group communication and involves:
    a. Atomicity: - ensures a message is delivered to all or none
    b. Ordered delivery: - ensures messages arrive to all in an order acceptable to the application (absolute, consistent and causal ordering techniques).
    c. Survivability: - guarantees correct message delivery in the face of partial failures of processes, machines or communication links replication plays a role.

vi. **Flexibility -** IPC primitives should be such that users have the flexibility to choose and specify the types and levels of reliability and correctness requirements of their application. etc. Also, IPC primitives must also have the flexibility to permit any kind of control flow between the cooperating processes, including synchronous and asynchronous *send/receive*.

vii. **Security -** end-to-end communication must not be interfered with. This may involve
   a. Authentication of the receiver(s) of a message by sender.
   b. Authentication of the sender of a message by its receiver(s)
   c. Message encryption before sending over the network

viii. **Portability -** exists in two forms.
   a. The message-passing system should be portable i.e., easy construction of a new IPC facility on another system by reuse of existing one.
   b. Applications on the existing message-passing system should be portable e.g., heterogeneity must be considered in design and high-level primitives provided to hide the heterogeneous nature of the network.

Issues in IPC by Message Passing

A message consists of a header (fixed length) and a variable-size collection of typed data objects.
The header has sender and receiver address; sequence number to manage data duplication and lost data, structural information that specifies the type of data and length.

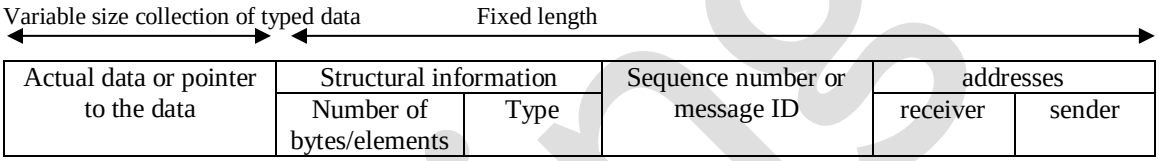


Fig. 2.1: Shape of a message

Type – refers to whether data to be passed is included within the message or the message contains only a pointer to the data, which is stored somewhere outside the contiguous portion of the message.
Size – refers to the length of the message.

The following issues need consideration in designing an IPC protocol for message-passing system.

- Who is the sender/receiver, is it one or many receivers? – authentication in 1:1, 1:M and M:N
- Is the message guaranteed to have been accepted by its receivers? – ack.
- Does sender need to wait for a reply? - Blocking and non-blocking.
- What happens if node crashes or communication link fails? – resend.
- What happens if receiver is not ready to accept message? - buffere
- If several messages are outstanding, can the receiver choose what order to receive them? (see pg 119) – absolute, consistent and causal ordering of messages.

These are addressed by the following methods/techniques.

1. **Synchronization**: Semantics used for this are classified as "blocking and "non-blocking" types. The non-blocking type does not block the execution of its invoker (i.e., control returns immediately, to the invoker) and vice versa. (Refer to the blocking and the non-blocking send and receive operations in OS1)

NB: Two methods exist in a non-blocking receive primitive for receiver to determine if a message arrives via buffer.

i. **Polling -** where receiver periodically checks the buffer status.
ii. **Interrupt -** software interrupt notifies the receiving process.

Thus, when both the "send" and "receive" primitives of a communication between two processes use blocking semantics, the communication is said to be synchronous, otherwise asynchronous

Synchronous communication is:

i. Easy and simple to implement
ii. Contributes to reliability
iii. But it limits the scope of concurrency and may bring about deadlocks.

NB: A flexible message-passing system should provide both blocking and non blocking primitives for "send" and "receive" for users to choose for their application.

2. Buffering: Synchronous and asynchronous communication corresponds respectively to "no buffer" and "buffer with unbounded capacity". The various forms of buffering include;

i. **Null buffer (No buffering): -** requires no buffer thus requires that
   - message stays in sender's process address space and "send" is delayed until receiver executes the corresponding "receive" i.e., sender process backed up and suspended in such a way that when it is unblocked, it starts by re-executing the send statement.
   - Message discarded and time out mechanism used to resend

ii. **Single message buffer: -** Null buffer is not suitable for synchronous communication and so single-buffer is used, that stores single message at a time. This is based on the fact that most systems under synchronous communication may have at most one message outstanding at a time.

iii. **Unbounded-capacity buffer**- suitable for asynchronous communication systems. Buffers all pending messages (but practically impossible)

iv. **Finite-bound (multiple-message) buffer**: - implemented for asynchronous communication since **iii** above is not practical. Incase of buffer overflow, two methods are applied:
   - Unsuccessful communication - message transfer simply fails and receiver notifies sender that buffer is full, but is less reliable.
   - Flow-controlled communication - sender is blocked until receiver accepts some message, but reintroduces synchronization that may create deadlocks and so not truly asynchronous.

3. Multidatagram messages: All networks have an upper bound on the size of data that can be transmitted at a time referred to as "max" transfer unit (MTU). On exceeding this, a message is broken into packets (datagrams) and so referred to as "multidatagram messages". Disassembly and reassembly are the works of the message-passing system.

4. Encoding and decoding of message data: It is impossible to reserve the structure of program objects from sender to receiver under heterogeneous systems to ensure data remains meaningful. It is still difficult even in homogenous systems because:

a. An absolute pointer value loses its meaning from one process address to another.
b. Different program objects occupy varying amounts of storage space e.g., long int, short int, etc i.e., responds to platforms that cannot respond to information uniformly e.g., heterogeneous platforms.

The above two problems call for conversion of such objects (coding/decoding) in some two ways.

- In tagged representation – the type of each program object and its value is encoded in the message.
- In untagged representation – the message data only contains program objects and so the receiving process must have a prior knowledge of how to decode the coded data.

5. Process Addressing: Message-passing system does support two types of process addressing i.e., naming of the parties involved in an interaction.

i. Explicit addressing – the process with which communication is desired is explicitly named as a parameter in the communications primitive used.

ii. Implicit addressing – e.g., sender names a service rather than a process e.g., in client server system, a receiver is willing to accept a message from any sender and vice versa. Deals with who is the sender and who is the receiver.

Primitives for explicit and implicit addressing of processes
Process addressing

i. `send (process-id, msg)` – send a message to the process identified by "process-id"

ii. `receive (process-id, msg)` – receive a message from the process identified by "process-id"

iii. `send-any (service -id, msg)` – send a msg to any process that provides the service of type "service-id"

iv. `receive-any (process-id, msg)` – receive a msg from any process and return the process from which the msg was received.

i and *ii* are explicit while *iii* and *iv* are implicit addressing.

6. Failure Handling: Failures occur due to node crash or communication link failure, leading to the following problems.

i. Loss of request message – due to link failure or receiver's node being down at the time of request.
ii. Loss of response message – due to link failure or sender's node being down at the time response message reaches.
iii. Unsuccessful execution of the request – due to receiver's node crashing while request is being processed.

To counter these problems, a reliable IPC protocol message-passing system is designed based on internal retransmission after timeouts and the return of acknowledgement.

7. Group Communication: Consists of the following schemes:

i. One-to-many
ii. Many-to-one
iii. Many-to-many

The following are issues related to the schemes: -

1. One-to-many communication: -
   i. Multiple receivers for a message from single sender.
   ii. Known as multiple communications (broadcast, which is a special form of multicast, sends to all nodes in a network).
   iii. The multicast/broadcast are important aspects of communication e.g., finding a service from a server or a CPU to offer a specific service.

Group Management: - May be of two types

- a "closed group": - only the members of that group may send message to the group, and an outside member cannot send a message to the group as a whole, but to only an individual of them.
- An "open group" – any system process may send message to the group as a whole.

A message-passing system with group communication facility provides the flexibility to create and delete groups dynamically and to allow a process to join or leave a group at any time. The mechanism to manage the groups and their membership information is implemented by a centralized "group server" where all requests to create, delete, add a member or remove a member from a group are sent. This method suffers from poor reliability and scalability of centralized systems. Replication of the servers is therefore necessary but not without the overhead of keeping the group information of all group servers consistent.

Group Addressing: - Addressed in two levels, the high level-name, as ASCII string, that is independent of the location information of the process in the group and the low-level, that addresses the hardware e.g., multicast/broadcast addresses.

In order to deliver a message to a receiver process: - the sender uses the high-level name, the kernel of the sending machine contacts the group server to obtain the low-level name of the group and the list of process identifiers of the process belonging to the group. Note that the sender simply sends a message to a group specifying its high-level name and the OS takes the responsibility to deliver the message to all the group members.

Buffered and Unbuffered Multicast – multicasting is asynchronous due to:

i. It is unrealistic to expect a sending process to wait until all the receiving processes that belong to the multicast groups are ready to receive the multicast message.
ii. The sending process may not be aware of all the receiving processes that belong to the multicast group.

Thus, in unbuffered multicast, message is lost and vice versa.

Send-to-All and Bulletin Board Semantics: -

i. **Send-to-all** – means that a copy of the message is sent to each process of the multicast group and the message is buffered until it is accepted by the process.
ii. **Bulletin-board semantics:** - the message is addressed to a channel, that plays the role of a bulletin board and receiving process takes their copies from here. It is more reliable than (1) because,
   - The relevance of a message to a particular receiver may depend on the receiver's state.

- Message not accepted within a certain time after transmission may no longer be useful, i.e., their value may depend on the sender's state.

Flexible Reliability in Multicast Communication: In 1:M communication, reliability is expressed in one of the following forms:

a. **The O-reliable**: - Where no response is expected from any of the receivers (e.g., in asynchronous multicast)
b. **The 1-reliable**: - sender expects a response from any of the receivers.
c. **The M-out-of-n reliable**: - the multicast group consists of n receivers and the sender expects a response from m ($1<m<n$) of the n receivers.
d. **All-reliable**-sender expects response from all the receivers of the multicast group e.g., when updating replicas of a file in a multicast.

2. Many-To-One-Communication

Is where the single receiver may be selective or non-selective. A selective receiver specifies a unique sender and they exchange messages only if that sender sends. But the non-selective receiver specifies a set of senders and a message exchange takes place once any one sender in the set sends.

3. Many-To-Many-Communication

This scheme takes care of 1:M and M:1 and in addition considers the issue of "ordered message delivery" in M: M. Ordered message delivery ensures that all messages are delivered to all receivers in an order acceptable to the application and thus requires sequencing. This can be executed in three ways: -

i. **Absolute ordering**: - Ensures that all messages are delivered to all receiver processes in the exact order in which they were sent. e.g., via the use of global timestamps as message identifiers (i.e., all clocks are synchronized). The kernel of each receiver machine saves all incoming messages in a separate queue and a sliding-window mechanism used to periodically deliver the message from the queue to the receiver (a fixed time interval used as window size). Messages whose timestamp values fall outside the window are left in the queue. The window size is also chosen taking into account the maximum possible time that may be required by a message to go from one machine to any other machine on the network.
ii. **Consistent ordering**: - Ensures that all messages are delivered to all receiver processes in the same order though this order may be different from the order in which messages were sent. (note that case (*i*) above requires globally synchronized clocks which are not easy to implement).

One way to implement consistent ordering is to make the M: N appear as a combination of M:1and 1:M schemes i.e., 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. Then the kernel of each receiver's machine saves all incoming messages meant for a receiver in a separate queue, such queued messages are delivered immediately to receiver unless there is a gap in the message identifiers causing messages after the gap not to be delivered until the ones in the gap have arrived. This method however suffers from a single-point of failure hence poor reliability.

Another way is a distributed algorithm that assigns a sequence number by distributed agreement among the group members and the sender. It runs thus:

a. Sender assigns a temporary sequence number to the message, and sends it to all the members of the multicast group, and it must be larger than the previously assigned sequence number, used by the sender
b. Each receiver returns a proposed sequence number to sender.
c. Sender selects the largest one proposed as the final sequence number for the message and sends it to all members in a commit message.
d. On receiving the commit message, each member attaches the final sequence number to the message.
e. Committed messages with final sequence number are delivered to the application programs in order of their final numbers.

iii. **Causal ordering**: - ensures that if the event of sending one message is causally related to the event of sending another message, then the two messages are delivered to all receivers in the correct order. But if they are not causally related, then the two messages may be delivered to the receivers in any order. Causal relationship implies that one message (2^{nd}) may be influenced by the (1^{st}).

CHAPTER 4: REMOTE PROCEDURE CALLS (RPC)

Introduction
An independently developed IPC protocol is tailored specifically to one application and so does not provide a foundation on which to build a variety of distributed applications. A need was therefore felt for a general IPC protocol that can be used for designing several distributed applications, and the RPC facility emerged out of this need.

RPC is a special case of the general message-passing model for IPC. Apart from the primary motivation for RPC in providing programmers with a familiar mechanism for building DS, it provides valuable communication mechanism suitable for building a fairly large number of distributed applications. Its popularity is due to the following features:

- Simple call syntax
- Familiar semantics i.e. due to its similarity to local procedure call (*lpc*)
- Its specification of a well-defined interface
- Its ease of use makes it easier to build distributed computations and to get them right
- Its generality
- Its efficiency i.e., procedure calls are simple enough for communication to be rapid
- It can be used as an IPC mechanism to communicate between processes on different machines as well as between processes on the same machine.

The RPC models
It is similar to the procedure call model used for the transfer of control and data within a program in the following manner:

- For making a procedure call, the call 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 the result.

RPC enables a call to be made to a procedure that does not reside in the address space of the calling process and the called process i.e., remote procedure, may be on the same computer or on different computer as the calling process. RPC facility uses a message-passing scheme for information exchange between the caller and the callee process.

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

RPC protocol makes no restrictions on the concurrency model implemented e.g., allowing only one process to be active at a time or allowing calls to be made asynchronously or having the server create a thread to process an incoming request so that the server can be free to receive other request etc. (what is a thread?)

Transparent RPC means that both local and remote procedures are "effectively" indistinguishable to programmers. This requires two types of transparencies

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

It is easy to achieve syntactic transparency since remote procedure calls are analogous to that of local procedure calls for most parts i.e.

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

But it is not easy to achieve semantic transparency due to:

i. Unlike local procedure calls, with remote procedure calls, the called process is executed in an address space that is disjoint from the calling program's address space and due to this reason; the called (remote) procedure cannot have access to any variables or data values in the calling program's environment.

ii. RPC are more vulnerable to failure than *lpc* since they involve two different processes and possibly a network and two different computers. This dictates that programs that make use of RPC must have the capability of handling even those errors that cannot occur in *lpc*.

iii. RPC consumes much more time (100 – 1000 times more) than *lpc* mainly due to the involvement of a communication network in RPCs. This automatically requires applications that use RPC to 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 prom programs, the interface to the underlying RPC system i.e., from the client and server processes. Also, 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. The implementation of an RPC mechanism therefore usually involves the following five elements of programs (Birel & Nelson 1984):

i. **The client -** I the client – is a user process that initiates a remote procedure call. It makes a perfectly normal call that invokes a corresponding procedure in the client.

ii. **The client stub** – responsible for
   a. Packing a specification of the target procedure and the arguments into a message and then asking the local RPCRuntime to send it to the server stub.
   b. On receipt of the result of procedure exception, it unpacks the result and passes it to the client.

iii. **The RPCRuntime** – handles transmissions of messages across the network between client and server machines. It is responsible for issues e.g., retransmissions, acknowledgements, packet routing and encryption.

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

v. **The server –** on receiving a call request from the server stub, the server executes appropriate procedure and returns the result of procedure execution to the server stub.

Nb: All the details of 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.

RPC messages

Based on the mode of client-server interaction, the two types of messages involved in the implementation of an RPC system are as follows

i. Call messages – that are sent by the client to the server for requesting execution of a particular remote procedure.

ii. Reply messages – that are sent by the server to the client for returning the result of remote procedure execution.

Server management

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

1. **Server implementation -** such servers can either be stateful or stateless. Stateful servers maintain client's information form one remote procedure call to the next which may help in subsequent executions. Stateless servers maintain no client state information.

   Why stateless servers

   They provide an easier programming paradigm because
   - They relieve the clients from the task of keeping track of state information
   - They are typically more efficient than stateful servers but the use of stateless servers is justified by the fact that they have a distinct advantage over stateful servers in the event of failure – they do not need to know that the server has crushed or that the network temporarily went down i.e., they make crash recovery easy.

Nb: In the case of stateful servers

- If a server crashes and then restarts, the state information that it was holding may be lost and the client process might continue its task unaware of the crash thereby producing inconsistent results.

- Also, when a client process crashes and then restarts its task, the server is left holding state information that is no longer valid but cannot be easily withdrawn – implying that the client of a stateful server must be properly designed to detect server crashes so that it can perform the necessary error handling activities.

2. Server creation semantics

Client and server processes are normally independent of each other. In this case server processes may either be created and installed before their client processes or be created on a demand basis. They may therefore be classified as

i. **Instance-per-call servers** – exists only for the duration of a single call. They are created by RPCRuntime on the server machine only when a call message arrives and is thereafter deleted. But it is associated with the following problems hence rarely used
   - They are stateless because they are killed as soon as they have serviced the call for which they were created, meaning that any state that has to be preserved across server call must be taken care of by either the client process or the supporting operating system.
   - When distributed applications need to successively invoke the same type of server several times, it is expensive since resource allocation and deallocation have to be done many times.

ii. **Instance-per-session servers** – exist for the entire session for which a client and server interact. A server of this type can retain useful state information between calls and so can present a cleaner more abstract interface to its clients. This services only a single client and hence only has to manage a single set of state information.

iii. **Persistent server** – generally remain in existence indefinitely and usually shared by many clients unlike the two cases above. They are usually created and installed before the clients that can use them. A persistent server can be bound to several clients and so the remote procedure that it offers must be designed so that interleaved or concurrent request from different clients do not interfere with each other.

Communication protocols for RPCs

Based on the needs of different systems, several communication protocols have been proposed for use in RPCs as outlined below

i. **The request protocol –** also known as the *R* protocol, used in the RPCs in which the called procedure has nothing to return as a result of procedure execution and the client requires no confirmation that the procedure has been executed. Only one message per call is transmitted i.e., from client to server and an RPC using the *R* protocol is called *asynchronous RPC.*

ii. **The request/reply protocol –** also known as the RR protocol. This protocol does not posses in its basic form, failure handling capabilities and therefore to take care of lost messages, the timeouts-and-retries technique is normally used along with the *RR* protocol.

iii. **The request/reply/acknowledge-reply protocol –** also known as the *RRA* protocol. The RRA protocol involves the transmission of three messages per call (two from client to server and one from server to client) in this protocol, there is a probability that the acknowledgement message itself may be lost. Its implementation therefore requires that the unique message identifiers associated with request messages must be ordered.

Some special types of RPC

i. **Callback RPC** – in the usual RPC protocol, the caller and the callee processes have a client-server relationship. The callback RPC on the other hand facilitates a peer-to-peer paradigm among the participating processes i.e., it allows a process to be both a client and a server.

ii. **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.

iii. **Batch mode RPC** – used to queue separate RPC request in a transmission buffer on the client side and then sends them over the network in one batch to the server.

RPC in heterogeneous environments

When designing an RPC system for heterogeneous environments, the three common types of heterogeneity that need to be considered are as follows

i. **Data representation –** an RPC system for 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.

ii. **Transport protocol –** for better portability of applications, an RPC system must be independent of the underlying network transport protocol.

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

Optimization for better performance

Some optimizations that may have significant payoff when adopted for designing RPC-based distributed applications include

i. Concurrent access to multiple servers
   - Use of threads
   - Use of early reply approach i.e., a call is split into two separate RPC calls, one passing parameters to server the other requesting the result.
   - Call buffering i.e., client does not interact directly with the server but through a buffer. Sends requests and deposits in buffer; server puts results in buffer, client goes ahead with other jobs and only polls buffer when needs results.

ii. Serving multiple requests simultaneously – use the approach of multithreaded server with dynamic threads creation facility for server implementation.

iii. Reducing per-call workload on servers – keep requests short and workloads low i.e., use stateless servers and let clients keep track of the progression of their requests sent to servers.

iv. Reply caching of idempotent remote procedures – caching strategy

v. Proper selection of timeout values – not too small/not too large. Some exponential back off strategy is good.

vi. Proper design of RPC protocol specification – to minimize the amount of data that has to be sent over the network and the frequency at which it is sent.

CHAPTER 5: RESOURCE MANAGEMENT

Defn: A resource manager schedules the processes in a DS to make use of the resources in such a manner that resource usage, response time, network congestion and scheduling overhead are optimized.

Introduction

A process may be migrated because: -

- node does not have required resources
- node has to be shut down
- if expected turnaround time will be better

It is the responsibility of the DOS to control the assignment of resources to processes and to route the process to suitable nodes of the system once a user submits a job.

In this chapter we will consider a resource to be a node/processor. A resource manager schedules processes and methodologies for scheduling can be broadly classified thus: -

i. **Task assignment approach** – where each process submitted by a user is viewed as a collection of related tasks which are then scheduled to suitable nodes to improve performance.
ii. **Load-balancing Approach** – all processes submitted are distributed to spread workload equally.
iii. **Load-sharing Approach** – attempts to ensure that no node is idle while processes want to be processed.

Desirable Features of a Good Global Scheduling Algorithm.

i. **No a priori knowledge about the processes -** A good scheduling algorithm should operate with absolutely no a priori knowledge about the process to be executed so as not to pose an extra burden on users to specify this information during job submissions.
ii. **Dynamic nature -** should be able to take care of the dynamically changing load (or status) of the various nodes of the system. (requires preemptive process migration facility)
iii. **Quick decision-making capability -** must make quick decision about the assignment of processes to processors. e.g., heuristic methods requiring less computational effort while providing near-optimal results are normally preferable to exhaustive (optimal) solution methods.
iv. **Balanced system performance and scheduling overhead -** algorithms that provide near-optimal system performance with a minimum of global state information gathering overhead are desirable (aging of information is bad).
v. **Stability -** an algorithm is unstable if it can enter a state in which all the nodes are spending all their time migrating processes without accomplishing useful work in an attempt to properly schedule the processes for better performance, termed as *processor thrashing.*
vi. **Scalability -** the algorithm should be capable of handling small as well as large networks e.g., probe only M of N nodes for selecting host.
vii. **Fault Tolerance -** should not be disabled by the crash of one or more nodes.
viii. **Fairness of service** – e.g., where some loads are larger than others, a node should be able to share some of its resources as long as its users are not significantly affected.

Explanations:

1. Task Assignment Approach: - Here, a process is considered to be made up of multiple tasks. The goal is to find an optimal assignment policy for the tasks of an individual process. With suitable assumptions, we seek to achieve: -

- Minimization of IPC costs
- Quick turnaround time
- High degree of parallelism
- Efficient utilization of system resources in general

But it lacks the dynamicity in changing work loads and it is deterministic.

The typical assumptions made in task assignment include:'

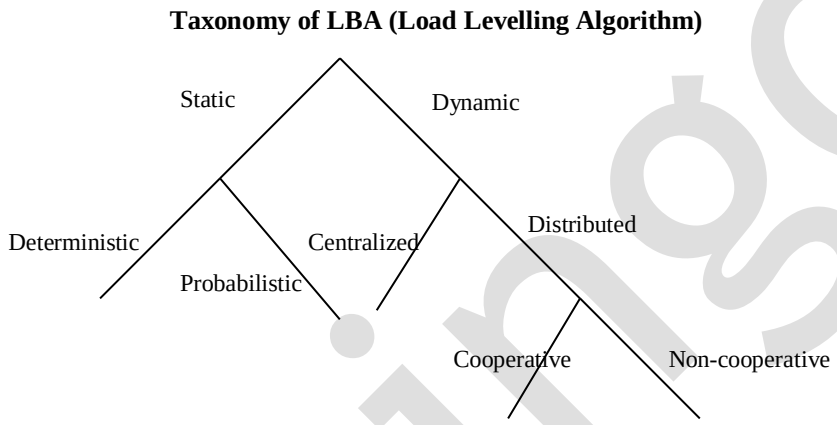
i. A process has been split into pieces called tasks along natural boundaries (*modular or logical chunking*) – each task has its own integrity and data transfers among tasks minimized.
ii. The amount of computation required by each task and the processors speeds are known – *deterministic*
iii. The cost of processing each task on every node of the system is known - *deterministic* .
iv. IPC costs between every pair of processes are known – considered to be zero if the processes are on the same node - *deterministic.*

v. Other constraints e.g., resource requirements by each task, the available resources at each node, precedence relationships among the tasks etc. are known.
vi. Tasks reassignment not possible – non-pre-emptive scheduling.

2. Load Balancing Approach: - Maximizes total system throughput. Various types of load-balancing algorithms include:

- **Static Vs Dynamic: -** Static algorithms use only information about the average behavior of the system, ignoring the current system state, while dynamic reacts to system state changes.

Fig: A taxonomy of load-balancing algorithms



Static is simple due to no need to maintain state information, but suffers from not being able to react to system state changes, while dynamic is vice versa.

- **Deterministic Vs probabilistic: -** Deterministic algorithms use information about the properties of the nodes and the characteristics of the processes to deterministically allocate processes to nodes. Probabilistic uses information regarding static attributes of the system e.g., number of nodes, processing capability of each node, network topology etc. to formulate simple process placement rules. But deterministic is difficult to optimize and is more expensive to implement while probabilistic is easier but has poor performance.

- **Centralized Vs Distributed: -** Decisions are centralized (one node) in centralized algorithms and distributed among the nodes on distributed. In centralized, it can effectively make process assignment decision because it knows both the load at each node and the number of processes needing service. Each node is responsible for updating the central server with its state information, but it suffers from poor reliability. Replication is thus necessary but with the added cost of maintaining information consistency. For distributed, there is no master entity i.e., each entity is responsible for making scheduling decisions for the processes of its own node, in either transferring local processes or accepting of remote processes.

- **Cooperative Vs non-cooperative: -** in non-cooperative, entities act autonomously, making scheduling decisions independently of others and vice versa for cooperative. But cooperative is more complex, leading to larger overheads, but displays better stability.

Issues in Designing Load-Balancing Algorithms.

The following should be considered: -

i. **Load estimation policy: -** that determines how to estimate the workload of a particular node of the system. Measurable parameters used may include:
   - Total number of processes on node at the time of load-estimation.
   - Resource demands of these processes
   - Instruction mixes of these processors – instruction sets and addressing modes
   - Architecture and the speed of the node's processor.

Since the above parameters may not necessarily work well, an acceptable method is to measure the CPU utilization of the nodes.

ii. **Process transfer policy: -** That determines whether to execute a process locally or remotely. This policy observes the "threshold policy" i.e., the limiting of a node's workload.

Two techniques, fig. a. single-threshold or fig. b. double-threshold can be used.

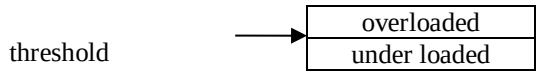


Fig a

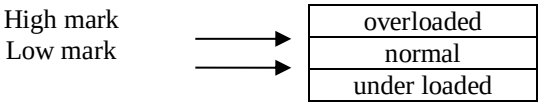


Fig b

In instance b,

- Overloaded region – new local processes sent for remote execution, new requests to accept remote processes are rejected.
- Normal region – new local processes run locally, request for remote processes rejected.
- Under-loaded region – new local processes run locally new remote processes accepted.

iii. **Location policies: -** Select the destination node for a process's execution, the main policies being:

a. **Threshold: -** selects destination node randomly and checks if the process transfer would place the node in a state that prohibits it from accepting remote processes.

b. **Shortest: -** a number of nodes chosen randomly and polled in turn to determine load and the one with least load given job.

c. **Bidding: -** each node takes on the role of "manager-contractor" where manager broadcasts and contactors bid. Best bidder wins but, in a case where contractor wins too much, it is bad. Hence the contractor has to send a message accepting/rejecting back to the manager, either for processes transfer or new broadcasting.

d. **Pairing: -** which pairs nodes and reduces the variance of loads only between the pairs. A node randomly chooses another and requests for pairing i.e., sharing load. At this instance, the requester node does not accept any communication from other nodes until it gets a response from the requested.

iv. **State info exchange policies: -** Proper selection of the state information exchange policy is essential, and may use one of the following policies for these purposes.

a. **Periodic broadcast** – each node broadcasts after every time *t*.
   - Not good due to heavy network traffic and possible existence of "fruitless" messages (i.e., messages from those nodes whose nodes' state have not changed within time *t*)
   - Scalability hence poor.

b. **Broadcast when state changes** – avoids problem of fruitless massages.
   - Small state changes are not necessarily reported to all nodes i.e., broadcasting occurs only when node switches load regions.
   - Works only with two-threshold transfer policy.

c. **On-demand exchange** – node broadcasts when either under/overloaded and only those nodes in need communicate. It works only with two-threshold transfer policy.

d. **Exchange by polling: -** an affected node searches for a suitable partner by randomly polling other nodes one by one. The polling process stops either when a suitable partner is found or predefined poll limit is reached e.g., m-out-of-n, where m<n.

v. **Priority Assignment Policies: -** Could take any of the three forms: -

a. Selfish – Local process have higher priority than remote but yields the worst response performance time among the three policies i.e., poor performance for remote but best response time for local processes. Beneficial for possesses arriving at a lightly-loaded node.

b. Altruistic: - remote processes have higher priority than local. Achieves best response time performance of the three policies. Remote processes experience lower delays than the local which is unfair.

c. Intermediate: - Plays between number of local and remote processes i.e., if the number of local processes > remote processes, local processes are given higher priority otherwise vice versa. Treats local processes better than remote and overall response time closer to (b) above.

vi. **Migration-Limiting Policies: -** Decides on the total number of times a process should be allowed to migrate and may use one of the following policies:

a. Uncontrolled: - Migration times unlimited but causes instability.
b. Controlled: - Sets a limit as to how many times a process may be migrated.

3. **Load-sharing Approach: -** Load balancing has been criticized for:

- The overhead involved in gathering state information is very large.
- It's not achievable because the number of processes in a node is always fluctuating.

Thus, it is necessary and sufficient to prevent some nodes from being idle while others are busy, which is the essence of sharing. Thus, policies for load sharing include:

a. **Load estimation policies: -** try only to check if a node is idle or busy, e.g., via measuring CPU utilization.
b. **Process transfer policies: -** May use single or double threshold policies i.e., in single, a node already executing one process may transfer others. In double however, anticipation is a factor i.e., nodes about to be underutilized are given more processes.
c. **Location Policies: -** which may take the forms of: -
   - *Sender-initiated policy* - in which sender node decides where to send process.
   - *Receiver-initiated policy* - in which receiver node decides from where to get the process.
d. **State information exchange policies:** A node normally exchanges state information only when its state changes that takes either of the forms below.
   i. Broadcast when state changes.
   ii. Poll when state changes.

CHAPTER 6: PROCESS MANAGEMENT

Is concerned with making the best possible use of the processing resources of the entire system by sharing them among all processes. Three important concepts are used to achieve this goal:

i. **Processor allocation: -** decides on which process is assigned to which processor.
ii. **Process migration: -** which is the movement of a process from its current allocation to the processor to which it can be assigned.
iii. **Threads: -** which deals with fine-grained parallelism for better utilization of the processing capability of the system.

Process migration

A process may be migrated before (non-preemptive) or after (pre-emptive) it starts executing. The latter is more costly since the process environment must also accompany the process. Migration involves:

i. Selecting a process to move.
ii. Selecting the destination node.
iii. Transferring the process to new node.

Desirable Features of a Good Process Migration Mechanism.

i. **Transparency: -** which occurs at two levels namely: -
   - *Object access level* - that allows access to objects e.g., files and devices to be done in a location-independent manner.
   - *System calls and interprocess communication level* - i.e., these must be location independent e.g., once a message is sent, it should reach its receiver process without need for resend if receiver moves to another node before it reaches.

ii. **Minimal interference: -** Migration must cause only minimal interference to the process of the process involved and to system as a whole e.g., by minimizing "freezing time".

iii. **Minimal Residual Dependencies: -** No residual dependency should remain on the previous node or else:
   - Migrated process continues with its load on previous node diminishing benefits of migration.
   - A failure/reboot of previous node causes process failure.

iv. **Efficiency: -** involves minimizing
   - Freezing time
   - Cost of locating an object – nearness, resource demands.
   - Cost of supporting remote execution once process is migrated – caching, replication.

v. **Robustness: -** i.e., failure of a node other than the one on which a process is currently running should not affect the accessibility/execution of that process.

vi. **Communication between co processes of a job: -** It is vital that co-processes communicate directly with each other irrespective of their locations.

Process Migration Mechanisms

Involves the following activities: - freezing the process, transferring the process, forwarding messages meant for the migrant process and handling communication between cooperating processes. Freezing a process means the execution of the process is suspended and all external interactions with the process are deferred. Freezing and restarting operations normally do differ from system to system, but some general issues involved in these operations are described as follows.

i. **Immediate and Delayed Blocking of the process: -** i.e., before a process is frozen, its execution is blocked either immediately or until the process reaches a suitable state for blocking. After blocking, it is advisable to wait for the completion of fast I/O operations (e.g., disk I/O) associated with the processes. Information about open files must be maintained as well. Finally, the process is restarted on its destination node in whatever state it was before migration.

ii. **Address Space Transfer Mechanisms: -** Migration involves transfer of the following information types from source to destination node.
   - Processes' state e.g., execution status, scheduling information, I/O states, objects accessible etc.
   - Process address space e.g., code, data and stack of the program.

The following are used as address transfer mechanisms: -

- *Total freezing*: - a process's execution is stopped while transferring its address space.
- *Pre-transferring*: - address space transferred while process still runs on source node.

- *Transfer on reference*: - assumes process only uses a small part of their address spaces while executing and so, the process address space is left at source node. As relocated process executes on destination node, a page of the migrant process's address space is transferred from its source to destination only when referenced.

Mechanisms for handling co processes:
Provide efficient communication between a process and its sub processes which might have been migrated on different nodes. Two mechanisms are:

i. Disallowing separation of co-processes: - This can be achieved via: -
   - Disallowing the migration of processes that wait for one or more of their children to complete.
   - Ensuring that when a parent process migrates, its children processes migrate along with it.

ii. Home node or Origin site concept – home node concept means that communication between a process and its sub-processes occurs via the home node. It allows the complete freedom of migrating a process or its sub processes independently and executing them on different nodes of the system. But message traffic and communication costs increase considerably since all communication between a parent process and its children processes take place via the home node.

Advantages of Process Migration
- **Reducing average response time of processes: -** via redistributing node loads to idle/less utilized nodes.
- **Speeding up individual jobs**: - Redistributing tasks to execute concurrently or using a faster CPU node or migrating job to a node with minimum turnaround time due to various reasons.
- **Gaining higher throughput: -** by application of a suitable load balancing policy.
- **Utilizing resources effectively**: - i.e., a process can be migrated to the most suitable node to utilize the system resources in the most effective manner.
- **Reducing network traffic: -** i.e., migrate a process closer to the resources it is using most heavily or migrate and cluster two or more processes which communicate quite often to same system node.
- **Improving system reliability: -** i.e., migrate critical process to a node of higher reliability, migrate a copy of critical process and execute both the original and copy concurrently or processes of a node may be migrated to another node before a dying node completely fails.
- **Improving systems security: -** i.e., running a sensitive process on a more secure node.

Threads:
May be referred to as lightweight processes that help in improving application performance through parallelism. In OS with threads facility, the basic unit of CPU utilization is a thread. A process, hence all its threads, are owned by a single user. Threads share the CPU just like processes on a time-sharing basis. The motivations for using threads are several. (*Read more about threads).*

CHAPTER 7: DISTRIBUTED FILE SYSTEMS (DFS)

A file is a named object that is explicitly created, immune to temporary failures and persists until explicitly destroyed. It has two main purposes: -

i. Permanent information storage.
ii. Information sharing

A file system is simply a subsystem of an operating system that performs file management activities e.g., organization, storing, retrieval, naming, sharing and protection of files. A distributed File System (DFS) normally supports the following: -

i. Remote information sharing – allows a file to be transparently accessed by the processes of any node of the system irrespective of the file's location.
ii. User mobility – user can work from different nodes at different times.
iii. Availability – files should be available even in the event of temporary failure of one or more nodes (fault tolerance) i.e., file replicas stored on different nodes.
iv. Diskless workstations – i.e., a DFS allows the use of diskless workstations.

A DFS provides the following three types of services: -
Storage service – allocation and management of space on secondary storage device.
True file service – concerned with the operations on individual files e.g., accessing, modifying, creating, deleting etc.
Name service – provides a mapping between text names for files and references to files i.e., file IDs. Also referred to as Directory Service.

Desirable Features of a Good DFS
A good DFS should display the following features: -

a) **Transparency** – which can be classified into four as: -
   i. **Structure transparency** – since a DFS normally uses multiple file servers, it should look to its clients like a conventional file system, offered by a centralized, time-sharing operating system.
   ii. **Access transparency** – both the local and remote files should be accessed in the same manner.
   iii. **Naming transparency** – a file name should give no hint as to where a file is located and file should move from one node to another without changing name.
   iv. **Replication transparency** – for a file replicated on many nodes, both the existence of multiple copies and their locations should be hidden from the clients.
b) **User mobility** – user should be free to work from different nodes at different times without performance loss (i.e., user's home directory automatically brought to user's environment).
c) **Performance** – the performance of a DFS should be comparable to that of a centralized system.
d) **Simplicity and ease of use** – i.e., semantics of the DFS should be easy to understand (i.e., good user interface and small number of commands), and also the file system should be able to support the whole range of applications.
e) **Scalability** – should easily cope with growth of nodes and users in the system, i.e., withstand high service load, accommodate user community growth and enable simple integration of added resources.
f) **High availability** – should continue to function even if partial failures occur due to e.g., communication link failure, a machine failure or a storage device crash.
g) **High reliability** – probability of loss of stored data should be minimized as far as is possible.
h) **Data integrity** – must be guaranteed.
i) **Security** – must be secure to assure users of the privacy of their data.
j) **Heterogeneity** - should allow a variety of workstations to participate in the sharing of files via a DFS, and should also allow the integration of a new type of workstation or storage media in a relatively simple manner.

File Models:
The two commonly used criteria for file modeling are structure and modifiability.

i. **Unstructured and structured files** – in the unstructured case, there is no substructure known to the file server and the contents of each of the file system appears to the file server as an uninterrupted sequence of bytes i.e., the interpretation of the meaning and structure of data stored in the files are entirely up to the application programs (Unix, MS DOS use this model).

Structured files (very rarely used nowadays) – a file appears to the file server as an ordered sequence of records. The unstructured model is liked because sharing of files by different applications is easier, i.e., different applications can interpret the contents of a file in different ways.

ii. **Mutable and immutable files:** Using the modifiability criteria, there are two file types – mutable and immutable. In the mutable type (used by most existing OS), an update performed on a file overwrites on its old content to produce the new contents.

In the immutable case (e.g., Cedar File Systems CFS) a file cannot be modified once created except to be deleted i.e., rather than updating the same file, a new version of the file is created each time a change is made to the file contents and the old version is retained unchanged. Immutable suffers from:

- Increased use of disk space.
- Increased disk allocation activity.

File Accessing Models

The file-accessing models of a distributed FS mainly depend on two factors i.e., the method used for accessing remote files and the unit of data access.

Accessing remote files:

i. **Remote service model** – here, the processing of the client's request for file access is performed at the server's node i.e., client's request goes to the server, server performs access request, result then forwarded back to client.

ii. **Data-caching model** – in this case, data requested is cached at client's node and processing performed at client's node. A replacement policy e.g., LRU is used to keep the cache size bounded. This method greatly reduces network traffic hence enhances performance and greater system scalability. But there is the problem of keeping cached data consistent on the various nodes and servers. There is the substantial overhead of write operation.

Unit of data transfer

Data may be transferred in various units during a single read or write operation.

i. **File-level Transfer Mode**: - A whole file is moved across the network between client and server. Advantages include: -

- Transmitting entire file is more efficient compared to page by page.
- Has better scalability because of fewer server accesses, resulting in reduced server load and network traffic.
- Disk access routines on the servers can be better optimized.
- Once an entire file is cached at a client's site, it becomes immune to server and network failure.
- It simplifies the task of supporting heterogeneous workstations (i.e., it is easier to transform an entire file for compatibility reasons).

But,

- Requires enough storage space at client's site.
- It is wasteful moving the whole file if only a small fraction is needed.

ii. **Block-level transfer model** – data is transferred in units of file blocks i.e., a contiguous portion of a file fixed in size. Advantages include:

- Does not require client to have large storage space.
- Eliminates the need to copy entire file when only a portion is needed, can thus be used in diskless workstations.

Disadvantages

- For full file access, multiple server requests are needed resulting in higher network traffic.
- Overall, it has poorer performance compared to (i) above since in most cases entire files are needed.

iii. **Byte-level transfer model** – units of bytes used as transfer:

- Provides maximum flexibility i.e., it allows storage and retrieval of an arbitrary sequential sub range of a file.

But,

- There is difficulty in cache management due to the variable data lengths for different access requests.

iv. **Record-level transfer model** – data transferred by records. Used in structured file models unlike the three above used in unstructured file models.

File Caching Schemes

A file caching scheme for a DFS should address the following key decisions (in addition to the centralized file systems that check on granularity of cached data, cache size and replacement policy).

i. **Cache location**: - is where the cache data is stored. Three possible locations of cache in a DFS are:

a. *Server's main memory* – eliminates disk access costs on a cache hit that results in a considerable performance gain as compared to no cache.

The decision to locate the cache here could be due to:

- It is easy to implement and it is totally transparent to the client.
- It is easy to always keep the original file and cached data consistent since both reside on the same node.
- Since a single server manages both the cached data and file, multiple accesses from different clients can be easily synchronized to support Unix-like file sharing semantics (once a change occurs on a file, it is availed immediately to requests thereafter).

But,

- Involves a network access for each file access operation by a remote client and processing of the access request by the server, hence it does not eliminate the network access cost and does not contribute to the scalability and reliability of the DFS.

b. *Client's disk*: - Eliminates network access but requires disk access cost on a cache hit.

Advantages

- Reliability i.e., data is always there.
- There is large storage capacity as compared to main memory cache.

But,

- This policy cannot work if the system is to support diskless workstations.

A disk access is required for each access request even when there is a cache hit hence results in considerable large access time.

c. *Client's main memory*: - eliminates both network access cost and disk access cost, thereby providing maximum performance gain on a cache hit.

- Permits workstations to be diskless.
- Contributes to scalability and reliability.

But,

- Not preferable to a client's disk cache when, large cache size and increased reliability of cached data are desired.

Modification propagation

This regards methods used to update caches: -

a. *Write-through scheme:* - When a cache entry is modified, the new value is immediately sent to the sever for updating the master copy of the file.

Advantage

- High degree of reliability and suitability for Unix-like semantics.

But,

- Its poor write performance i.e., each write access has to wait until the information is written to the master copy of the server. Good only for read accesses than write accesses.

b. *Delayed-write scheme:* - When a cache entry is modified, the new value is written only to the cache and the client just makes a note that the cache entry has been updated. Later, all updated cache entries corresponding to a file are gathered together and sent to the server at a time. Uses three approaches namely: -

- **Write on ejection from cache**: - Modified data in a cache is sent to the server when the cache replacement policy has decided to eject it from the client cache.
- **Periodic write**: - Cache scanned periodically at regular intervals, and any cached data that have been modified since the last scan is sent to the server.
- **Write on close**: - Modification made to a cached data by a client is sent to the server when the corresponding file is closed by the client.

Cache validation schemes

Two approaches to verify the validity of cached data include: -

i. **Client initiated approach: -** Client contact server and checks whether its locally cached data is consistent with the master copy. May apply the following methods: -
   a. Checking before every access: - Suitable for supporting Unix-like semantics (server has to be contacted on every access and defeats the purpose of caching).
   b. Periodic checking: - Intervals of time.
   c. Check on file open: - Client cache entry validated only at this time (i.e., clients cache entry is validated only when the client opens the corresponding file for use). Suitable for supporting session semantics.

ii. **Server initiated approach: -** Used when the frequency of validity check is high. Client informs server when opening files (indicates whether reading, writing or both). Server keeps records of client's states and monitors file usage modes.

Problems.

   a. Violates the role of servers simply responding to client's request.
   b. Requires file servers to be stateful.
   c. A check-on-open, client-initiated cache validation approach must still be used along with the server-initiated approach. (since some other client may have modified the file after another client requiring re-use had closed it).

File replication

File replication is the primary mechanism for improving file availability (replicas are associated with file servers while caches are associated with client nodes).

Advantages

- *Increased availability*: - Masks and tolerates failures in the network gracefully.
- *Increased reliability*: - Existence of multiple copies of files
- Improved response time: - Enables data to be accessed either locally or from a node to which access time is lower than the primary copy access time.
- *Reduced network traffic*: - Client's access request serviced locally for file server that resides on a client's node.
- *Improved system throughput*: - Several client's requests serviced in parallel by different servers.
- *Better scalability*: - Due to workload distribution.
- *Autonomous operations*: - Temporary autonomous operations on client machines may be achieved via replication on the file server residing at the client's node.

Fault tolerance

The primary file properties that directly influence the ability of a DFS to tolerate faults are: -

- *Availability:* - Refers to the fraction of time for which the file is available for use.
- *Robustness:* - Refers to the power of a file to survive crashes of the storage device and decays of the storage medium on which it is stored.
- *Recoverability*: - Refers to the ability to be rolled back to an earlier consistent state when the operation on the file fails or is aborted by the client.

Note:

   c. *Stateful file servers*: - Maintain clients' state information from one access request to another.
   d. *Stateless file servers*: - Do not maintain any client state information.

CHAPTER 8: SECURITY

Users will trust their systems only if the resources/information are protected against destruction/unauthorized access. Irrespective of the operation environment, some common goals of computer security include:

- **Secrecy** - information accessible to only authorized users
- **Privacy** - misuse of information must be prevented.
- **Authenticity** - verification of authenticity necessary.
- **Integrity** - information protected against accidental destruction or intentional corruption by the unauthorized.

However, a total approach to computer security involves both external and internal security.

a. External security – secures the system against external factors e.g., fires, floods, stealing, leakage etc.
b. Internal security – deals mainly with user authentication and access control i.e., beyond authentication, you need to access only what you require to complete your task (need-to-know principle or principle of least privilege).
c. Communication security – ensures that data is not interfered with while in transit.

Nb: Providing both internal and external security in DS is more difficult due to lack of a single point of control and insecure networks.

Potential attacks to computer systems

An intruder may introduce two forms of attacks namely:

a. Passive attacks – do not cause any harm to the system being threatened. Involves stealing unauthorized information from the system without interfering with its normal function. May include the following methods
    - Browsing – reading of stored files, packets in transit etc. Requires access control.
    - Leaking – intruder uses a legitimate user to access information.
    - Inferencing – intruder tries to draw some inference by closely observing/analyzing the systems data/activities carried out by the system.
    - Masquerading – intruder masquerades as authorized user to gain access.
b. Active attacks – interfere with damaging effects, e.g., corrupting files, destroying data, imitating hardware errors, slowing down system, filling up memory, system crushing etc. Some commonly used forms of attack include
    - Viruses – a piece of code attached to a legitimate program that when executed, infects other programs in the system by replicating and attaching itself to them. It is not a complete program.
    - Worm – programs that spread from one computer to another in a network of computers. It is a complete program.
    - Logic bombs – are dormant programs waiting for a trigger condition to explode. It destroys data and spoils systems software of the host computer.

Cryptography

Protects private information against unauthorized access where it may be difficult to provide physical security. It lies on the premise that it is not possible to prevent copying but it is better to prevent comprehension.

Case studies

Closely examine some of the DS that are currently either in commercial use or still under laboratory conditions.

-- The end --