Module 1

Module 3

Chapter 3 : Synchronization 3-1 to 3-28

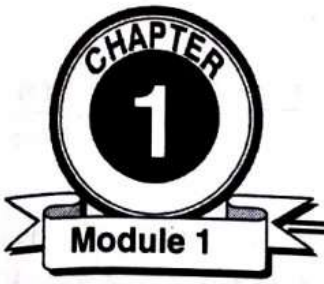
Module 4

Chapter 4 : Resource and Process Management 4-1 to 4-27

Module 5

Module 6

❑❑❑



Introduction to Distributed System

Syllabus

Characterization of Distributed Systems: Issues, Goals, and Types of distributed systems, Distributed System Models, Hardware concepts, Software Concept.

Middleware: Models of Middleware, Services offered by middleware, Client Server model.

1.1 Introduction

- The development of powerful microprocessors and invention of the high speed networks are the two major developments in computer technology. Many machines in the same organization can be connected together through local area network and information can be transferred between machines in a very small amount of time.
- As a result of these developments, it became easy and practicable to organize computing system comprising large number of machines connected by high speed networks. Over the period of last thirty years, the price of microprocessors and communications technology has constantly reduced in real terms.
- Because of this, the distributed computer systems appeared as a practical substitute to uniprocessor and centralized systems. The networks of computers are present all over.
- Internet is composed of many networks. All these networks separately and in combination as well, share the necessary characteristics that make them pertinent topics to focus under distributed system.
- In distributed system, components located on different computers in network communicate and coordinate by passing messages. As per this argument, following are the characteristics of the distributed system.
  - Internet
  - Intranet, which is small part of internet managed by individual organization.
  - Mobile and ubiquitous computing.
- Resources required for computation are not available on single computer. These resources which are distributed among many machines can be utilized for computation and it is main motivation behind distributed computing. Followings are some of the important issues that need to be considered.

1. Concurrency

- The different cooperating applications should run in parallel and coordinate by utilizing the resources which are available at different machines in network.

2. Global Clock not Available

- Cooperation among applications running on different machines in network is achieved by exchanging the messages. For cooperation, applications action at particular time is exchanged. But, it is not easy to have all the machines clocks with same time.
- It is difficult to synchronize the different machines clock in network. Therefore, it is necessary to take into consideration that single global clock is not available and different machines in LAN, MAN or WAN have different clock time.

3. Independency in Failure of System Component

- Any individual component failure should not affect the computation. It is obvious that, any software or hardware component of the system may fail.
- This failure should not affect the system from running and system should take appropriate action for recovery.

Definition of Distributed System

- A computer network is defined as a set of communicating devices that are connected together by communication links. These devices include computers, printers and other devices capable of sending and/or receiving information from other devices on the network. These devices often called as node in the network. So computer network is interconnected set of autonomous computers.
- A distributed system is defined as set of autonomous computers that appears to its users as a single coherent system. Users of distributed system feel that, they are working with a single system.
- Following are the main characteristics of distributed system.
  - A distributed system comprises computers with distinct architecture and data representation. These dissimilarities and the ways all these machines communicate are hidden from users.
  - The manner in which distributed system is organized internally is also hidden from the users of the distributed system.
  - The interaction of users and applications with distributed system is in consistent and identical way, in spite of where and when interaction occurs.
  - A distributed system should allow for scaling it.
  - Distributed system should support for availability. It should be always available to the users and applications in spite of failures.
  - Failure handling should be hidden from users and applications.
- Network contains computers with different architectures (heterogeneous). To offer the single system view of distributed system in this heterogeneous environment, a middleware layer is often placed between higher layer that comprises of user and applications and lower layer comprising the operating system and communication facilities.

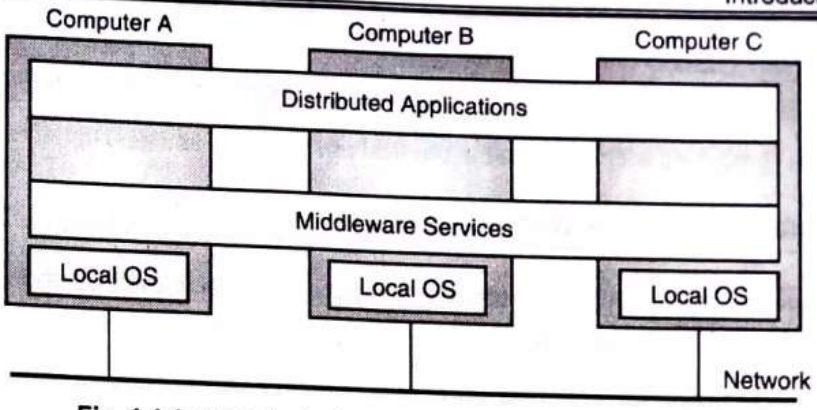


Fig. 1.1.1 : Distributed system organized as middleware

In Fig. 1.1.1, applications are running on three different machines A, B and C in network.

1.2 Examples of Distributed System

1.2.1 Internet

- Applications running on different machines communicate with each other by passing messages. This communication is possible due to different protocols designed for the same purpose.
- Internet is a very large distributed system. User at any location in network can use services of World Wide Web, email, file transfer etc. These services can be extended by addition of servers and new types of services.
- Internet Service Provider (ISP) provides internet connection to individual users and small organizations. Email and web hosting like local services are provided by ISP and it also enable the users to access services available at different locations in internet.

1.2.2 Intranet

- Intranet is small part of internet belonging to some organization and separately administrated to enforce local security policies. Intranet may comprise many LANs which are linked by backbone connection.
- Intranet can be administrated by administrator of single organization and its administration scope may vary ranging from LAN on single site connected to a connected.
- LANs that is part of many braches of companies. Router connects intranet to internet and user in intranet then can use services available in rest of the internet.
- Firewall which filters incoming and outgoing messages protects unauthorized messages to enter or leave the intranet.

1.2.3 Mobile and Ubiquitous Computing

- Many small and portable devices manufactured with advancement in technology is now integrated into distributed system. Examples of these devices are Laptop, PDAs, Mobile phones, Cameras, pagers, smart watches, devices embedded in appliances, cars and refrigerators etc.
- As these devices are portable and easily gets connected, mobile computing became possible. Mobile users can easily use resources available in network and services available in internet.

- Ubiquitous computing ties together the many small chip devices in available physical environment of home, office or elsewhere. In mobile computing, user location can be anywhere but can use resources available everywhere.
- Whereas, in ubiquitous computing users are in same physical environment and gets benefited.

1.3 Resource Sharing and the Web

- Hardware resources such as printers, disks are shared to reduce cost. Users can also share web pages or databases. The pattern of sharing in such large environment of distributed system cannot be same.
- All users can share search engine for searching purpose or small group of users can share their files or document among them. The resources available at different locations in large network can be used by users or applications in it.
- These resources are treated as services and present its functionality to users and applications. For example, shared files can be accessed by using file service. Sharing of resources available on different computers of large size system cannot be possible without communication.
- Thus, resource to be shared must be managed by some program which should provide communication interface which allows effective access and updating of this resource with reliability and in consistent way.
- Server application provides services. Client application running on other machine, request the service of server by sending message. In this scenario, we say that client is invoking an operation on server.
- Server processes the request and sends back response to client. This is called remote invocation by client.
- Requesting application is client and responding application is server. Server may request the service of other server also. Here, client and servers are processes and not computers.

1.3.1 World Wide Web (WWW)

- This system publishes, helps in accessing resources and services available in internet. Internet user uses web browser application to retrieve documents listen audio or view video streams.
- In any document, hyperlink is available to organize, view the contents or can be used as references to other documents and resources which are already available in web.
- Thus, users may go indefinite way to view required information as hyperlinks are provided in each viewed document for further information of their use. Web is open to extend, add new services and open to implement in new way with assurance about not affecting the existing system. For example, any browser can access any web server.
- This flexibility of web is applicable for all types of contents to publish. If new format or type of content is invented by any user then web can publish it. For this new content presentation, plug-ins in browser can be added.
- Webs working is based on three basic components which are HTML (Hypertext marks up language), HTTP (Hypertext transfer protocol) and URLs (Uniform resource locators).
  1. **HTML :** Content of web page like text and image is presented by using HTML. These structured items in web page are organized as headings, paragraphs, tables and images. Links are also provided to access the resources present at other location in network.
  2. **HTTP :** Web browser or client applications use this protocol to interact with web servers. Following are main features of HTTP.

- **Request-reply Interaction :** Client send request to server using URL of the resource to be accessed.
- **Content Types :** The client sends request to server which the returns response to client. Browsers are not necessarily capable to process this received content from server. Server mentions content type in reply message so that browser can process it. Browser may take other applications help to process the content if it is not capable to process it.
- **Resource access :** Browser sends many requests concurrently to server to minimize the delay to the user.
- **Access Control :** Access to particular resource can be restricted for unauthorized users. Users should be authenticated to use this resource.

3. **URLs :** Browser uses URL to locate the resource on web server. URL contains two components. These are scheme and scheme-specific-location. The first component declares type of URL. For example, email address, URL to access the file using FTP protocol, nntp to specify news group and telnet for remote login are the examples of scheme.

1.4 Issues and Goals

Following are the issues and goals related to design of distributed system.

1.4.1 Heterogeneity

- We cannot have homogeneous environment in wide area network. The types of networks, architecture of the different computers in different networks, operating system running on different computers, programming languages used and implementation of products by different developers cannot be expected to be same. Internet enables users to access services and run applications in heterogeneous environment.
- There are different implementations of internet protocols for different type of networks. If the machines architecture is different then data representation supported by architecture is also different. Hence, different computers in distributed system have different representation for integer, characters and other data items. For example, IBM mainframe uses EBCDIC character code and IBM PC uses ASCII.
- It is necessary to deal with such differences in data representation when messages need to be exchanged between applications running on such different hardware. Different programming languages also use different data representations for data structures such as array and records.
- Applications developed in different languages should be able to communicate with each other. This issue also needs to be addressed. It is also necessary that products developed by different developers should communicate with each other. This is for example for network communication and representation of primitive data items and data structures in messages.
- Internet protocols have these agreed and adopted standards. In future, design of distributed system, these standards should be developed and adopted for communication among heterogeneous hardware and software components.

1.4.2 Making Resources Accessible

- Users and applications should be able to access the remote resources in easy way. Distributed system should allow sharing these remote resources in efficient and controlled manner.

- Resource sharing offers saving in cost. One printer can be shared among many users in office instead of having one printer to each individual user. There can be more saving in cost if expensive resources are shared.
- By connecting users and resources, it becomes easier to work together and exchange information. The success of the Internet is due to its straightforward protocols for exchanging files, mail, documents, audio, and video. The worldwide spread people can work together by means of groupware. Electronic commerce permits us to purchase and sell verity of goods without going to shop or even leaving home.
- The increase in connectivity and sharing also increases the security risk and to deal with it is equally important. Presently, systems offer fewer defenses against eavesdropping or intrusion on communication.
- A communication can be tracked to construct a favorite profile of a particular user. This clearly violates privacy, particularly if it is done without informing the user. A allied problem with increased connectivity can also cause unnecessary communication, for example electronic junk mail, called as spam. Special information filters can be used to select inward messages based on their content.

1.4.3 Distribution Transparency

The second main goal of a distributed system is to hide the actuality of physical distribution of processes and resources across several computers. A transparent distributed system offers its feel to users and applications as a single computer system. Following types of transparency is present in distributed system.

- **Access Transparency :** It hides the dissimilarities of data representation of different machines in the network and the manner in which remote resources are accessed by the user. Intel machines use little endian format to order bytes and SPARC uses big endian format. So Intel machines transfer high order bytes in beginning and in case of SPARC low order bytes are transmitted first. Different operating systems also have their own file name convention. All these dissimilarities should be hidden from users and applications.
- **Location Transparency :** This transparency hides the location of the resources in distributed system. For example, URL used to access web server and file does not give any idea about its location. Also name of the resource remains same although it changes the location when moved between machines in the distributed system.
- **Migration Transparency :** It hides the fact that resources are moved from one location to other. It does not affect the way in which these resources are accessed. Processes or files often are migrated by system to improve the performance. All this should remain the hidden from user of the distributed system.
- **Relocation Transparency :** If the user or application is using the resource and during use of it is if moved to other location then it remains hidden from user. For example if user is traveling in car and changing the location frequently still he or she continuously uses the laptop without getting disconnected.
- **Replication Transparency :** It hides the fact that many copies of the resource are placed at different locations. Often resources are replicated to achieve availability or placed its copy near to location of its access. This activity of the replication should be transparent to the user.
- **Concurrency Transparency :** It hides the sharing of the resource by many users of the distributed system. If one user is using the resource then other should remain unknown about it. Many users can have stored their files on file server. It is essential that each user does not become aware of the fact that other is using the same resource. This is called concurrency transparency.

- **Failure Transparency :** It hides the failure and recovery of the resource from user. User does not become aware of failure of resource to work appropriately, and that the system then recovers from that failure. It is not possible to achieve complete failure transparency. Because for example, if network fails user can notice this failure.
- **Performance Transparency :** In order to improve performance system automatically takes some action and it user remains unaware of this action taken by system. In case of overloading of processor jobs gets transfer to lightly loaded machine. If many requests are coming from users for a particular resource then resource gets placed to nearest server of those users.
- **Scaling Transparency :** The main goal of this transparency is to permit the system to expand in scale without disturbing the operations and activities of existing users of distributed system.

1.4.4 Scalability

- If system adapt to increased service load then it is said to be a scalable system. There are three ways to measure the scalability of the system.

(i) Scalable with respect to its size
(ii) Geographically scalable
(iii) Administratively scalable

- Practically, scalable system leads to some loss of performance as the system scales up. Following are the problems related with scalability.

(i) Scalable with respect to its size

- System easily adapts addition of more users and resources to the system.
- In order to supports for more number of users and resources, it is necessary to deal with centralized services, data and algorithms. If centralized server is used to provide services to more number of users and applications then server will become bottleneck.
- In spite of having more processing power, large storage capacity with centralized server, more number of requests will be coming to server and increased communication will restrict further growth.
- Like the centralized services, centralized data also is not good idea. Keeping a single database would certainly saturate all the incoming and outgoing communication lines.
- In the same way above, centralized algorithms also a not good idea. If centralized routing algorithm is used, it will collect the information about load on machines and communication lines. The same information algorithm processes to compute optimal routes and distribute to all machines. Collecting such information and sending processed information will also overload the part of network. Therefore use of such centralized algorithms must be avoided. Preference should be given to only decentralized algorithms.

(ii) Geographically scalable

- Although users and resources may lie far distance apart geographically, still system allows the users to use resources and system.

- In synchronous communication client blocks until reply comes from server. Distributed system designed for local area networks (LANs) are based on synchronous communication due to small distances between machines.
- Therefore it is hard to scale such systems. A care should be taken while designing interactive applications using synchronous communication in wide area systems as machines are far apart and time required for communication is three orders magnitude slower with compare to LAN.
- A further problem that get in the way of geographical scalability is that communication in wide-area networks is intrinsically unreliable, and virtually always point-to-point. On the contrary, local-area networks generally offer highly reliable communication facilities based on broadcasting, which ease development of the distributed systems.
- If system has several centralized components then geographical scalability will be restricted because of the performance and reliability problems resulting from wide-area communication. Use of centralized components will lead to waste of network resources.

(iii) Administratively Scalable

- Although system spans many independent administrative organizations, still its management is easy.
- In order to scale a distributed system across multiple, independent administrative domains, a key problem that requires to be resolved is that of conflicting policies with respect to resource usage (and payment), management and security.
- Distributed system components residing in one domain can always be trusted by users that work within that same domain. In this scenario, system administration may have tested and authorized applications, and may have taken special measures to make sure that such components cannot be tampered with. Here, the users trust their system administrators. But, this trust does not cross domain boundaries naturally.
- If a distributed system crosses the domain boundaries, two forms of security measures require to be taken. First is, the distributed system on its own must protect against malicious attacks from the other new domain. Second, the other new domain on its own must protect against malicious attacks from the distributed system.

Scaling Techniques

Following are the three techniques for scaling :

1. Hiding communication latencies
2. Distribution
3. Replication

delay period, hiddenness

1. Hiding communication latencies

- Geographical scalability can be achieved by hiding communication latencies. In this case, waiting by client for response from geographically remote server can be avoided by using asynchronous communication for implementation of requester's application. In asynchronous communication instead of blocking client carry out other work until response is received.
- Batch-processing systems and parallel applications make use of asynchronous communication, in which if one task is waiting for communication to finish, some independent tasks can be scheduled for execution.

- On the other hand, a new thread of control can be started to carry out the request. Even though it blocks waiting for the reply, other threads in the process can carry on.
- In case of interactive applications, asynchronous communication cannot be used effectively. In this case to minimize the communication cost, server side computation of the request can be moved to the client side. Client side validation of form is the best example of this approach. Instead of carrying out validation at server side, it is better to shift it at client side.

2. Distribution

- Distribution is important scaling technique. In distribution component of the system divided into smaller parts, and then kept those parts across the system by spreading it. A good example of distribution is the Internet Domain Name System (DNS).
- There is hierarchical organization of DNS name space into a tree of **domains.** These domains are divided into non-overlapping **zones.** The names belonging to each zone are managed by a single name server.

3. Replication

- Although problems related to scalability degrade the performance, it is in general a good thought in fact to **replicate** components across a distributed system. Apart from increasing availability of the component, replication also balances the load between components which result in achieving the better performance. In WAN based system, placing a copy close to accessing location hides much of the communication latency problems described above.
- Similar to replication, caching leads to make a copy of a resource in the near to the client accessing that resource. But, contrary to replication, caching decision is taken by the client of a resource instead of owner of a resource.
- Caching is carried out on demand while replication is often planned in advance. The disadvantage of caching and replication that may have negative effect on scalability. Since multiple copies of a resource exist, making change in one copy makes that copy different from the others. As a result, caching and replication leads to consistency problems.

.4.5 Openness

Openness is the important goal of the distributed system. An **open distributed** system provides services as per standard rules that tell the syntax and semantics of those services. As, in computer networks, standard rules state the message format, its contents and meaning of sent and received messages. All these rules are present in protocols.

Similar to above, in distributed systems, services are usually specified through interfaces, which are expressed in an Interface Definition Language (IDL). The definitions of the interfaces written in an IDL almost capture only the syntax of these services.

They state exactly the names of the available functions together with parameters type; return values, exceptions likely to be raised etc. The semantics of interfaces means specification of what these interfaces can perform. Actually, these specifications are specified in an informal way through natural language.

Processes make use of interfaces to communicate with each other. There can be different implementation of these interfaces leading to different distributed system that function in the same manner.

- Appropriate specifications are complete and neutral. Complete specifications specify the whole thing that is essential to make an implementation. But, many interface definitions does not obey completeness.
- So that it is required for a developer to put in implementation-specific details. If specifications do not impose what an implementation should look like : they should be neutral. Completeness and neutrality are significant for interoperability and portability.
- An open distributed system should allow configuring the system out of different components from different developers. Also, it should be trouble-free to put in new components or replace existing ones without disturbing those components that stay in place. It means, an open distributed system should also be extensible. Open system interfaces are published.
- Monolithic approach should be avoided for building the system. There should be separation between policy and mechanism. For example, apart from storing the documents by browser, users should be able to make a decision which documents are stored and for how much duration. User should be able to set it dynamically.
- User should be able to implement his own policy as a component that can be plugged into the browser. Certainly implemented component must have an interface that the browser can recognize so that it can call procedures of that interface.
- Open distributed system provides uniform communication mechanism and it is also based on published interfaces in order to access the common resources. It also considers heterogeneous environment in terms of hardware and software.

1.5 Types of Distributed Systems

- There are a different distributed computing systems
  1. Distributed Information Systems
  2. Distributed Embedded Systems
- The following discussion describes these systems.

1.5.1 Distributed Computing Systems

- One class of distributed system is intended for high-performance computing tasks. Cluster computing comprises the hardware that consists of a set of similar workstations or PCs running same operating systems and closely connected through a high speed local-area network.
- In case of grid computing, distributed systems are built as a group of computer systems. Here administrative domain of each system may be distinct, and may be very dissimilar in hardware, software, and deployed network technology.

1.5.1(A) Cluster Computing Systems

- Cluster computing systems were accepted because of reduced cost and improved performance of computers and workstations. From technical and cost point of view, it became reliable to build supercomputer using off-the-shelf technology by simply connecting a set of comparatively simple computers in a high-speed network.

- In almost all cases, cluster computing is used for parallel programming in which a single program is run in parallel on several machines.
- The common configuration of Linux-based Beowulf clusters is shown in Fig. 1.5.1. Each cluster comprises a set of compute nodes. All these compute nodes are controlled and accessed through a single master node.
- The master node characteristically manages the allocation of nodes to a specific parallel program, holds a batch queue of submitted jobs, and offers an interface for the users of the system.
- The master in fact runs the middleware required for the execution of programs and management of the cluster, whereas the compute nodes requires only a standard operating system. Libraries offer advanced message based communication facilities.

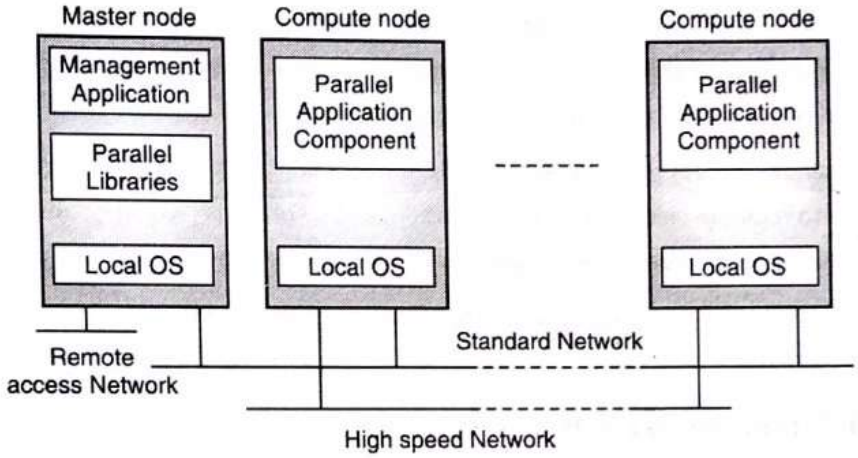


Fig. 1.5.1 : Cluster computing system

1.5.1(B) Grid Computing Systems

- In contrast to homogeneous environment of cluster computing systems, grid computing systems includes a high degree of heterogeneity. In this case hardware, operating systems, networks, administrative domains, security policies are different.
- In a grid computing system, resources from distinct associations are brought together to let the collaboration of a group of people or institutions. This collaboration leads to virtual organization. The people from same virtual organization are given access rights to the resources that are offered to that organization.
- These resources comprise compute servers (together with supercomputers, probably implemented as cluster computers), storage facilities, and databases. Also, special networked devices like telescopes, sensors, etc., can be made available as well. A layered architecture for grid computing systems is shown in following Fig. 1.5.2.

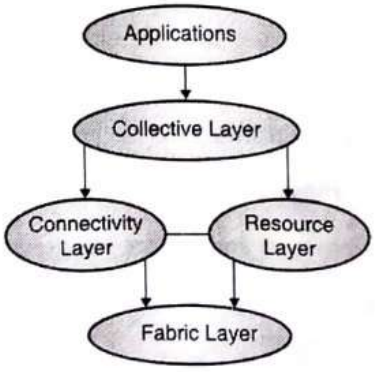


Fig. 1.5.2 : A layered architecture for grid computing systems

- **Fabric Layer :** It is a lowest layer and offers interfaces to local resources at a specific site. These interfaces are customized to permit sharing of resources within a virtual organization.
- **Connectivity layer :** This layer contains communication protocols to offer support for grid transactions that span the usage of multiple resources. Protocols are required to send data among resources or to access a resource from a distant location. Also, this layer will have security protocols to authenticate users and resources. Instead of user, if users program is authenticated then the handover of rights from user to program is carried out by connectivity layer.
- **Resource layer :** This layer manages a single resource. It makes use of functions offered by the connectivity layer and calls straight way the interfaces made available by the fabric layer. As an example, this layer will provide functions for getting configuration information on a particular resource or generally, to carry out specific operations such as process creation or reading data. Hence this layer is responsible for access control, and hence will be dependent on the authentication carried out as part of the connectivity layer.
- **Collective Layer :** It handles access to multiple resources and contains services for resource discovery, allocation and scheduling of tasks onto multiple resources, data replication, and so on. This layer contains many diverse protocols for verity of functions, reflecting the broad range of services it may provide to a virtual organization.
- **Application Layer :** This layer contains applications that functions within a virtual organization and which utilizes the grid computing environment.

1.5.2 Distributed Information Systems

- Many organizations deal with networked applications facing the problems in interoperability. Many existing middleware solutions provide easy way to integrate applications into an enterprise-wide information system.
- In most of the cases networked application is a client server application in which server run the application (database) and clients sends request to it. After processing request server sends reply to client.
- If integration is done at lowest level, it would permit clients to enclose a many requests, perhaps for different servers, into a single larger request and let it be executed as a distributed transaction. The basic scheme was that either all or none of the requests would be executed.
- Such integration should also happen by allowing applications to communicate straight way with each other. This has shown the way to enterprise application integration (EAI).Above two forms of distributed systems is explained below.

1.5.2(A) Transaction Processing Systems

Programming using transactions involves primitives that are either provided by underlying distributed system or by the language runtime system. Following are the examples of primitives for transactions.

- **BEGIN_TRANSACTION :** For marking the start of transaction.
- **END_ TRANSACTION :** For terminating the transaction and try to commit.
- **ABORT_ TRANSACTION :** For killing the transaction and restore the previous state.
- **READ :** To read the data from table or file.
- **WRITE :** To write the data to table or file.

The main property of a transaction is either all of its operations are executed or none are executed. The operations that are present in the body of transaction can be system calls, library procedures, or programming language statements. Following are the ACID properties of transaction.

1. **Atomic :** Execution of transaction is individual.
2. **Consistent :** There is no violation of system invariants by transaction.
3. **Isolated :** Concurrent transactions do not interfere with each other.
4. **Durable :** After commit is done by transaction, the changes are permanent.

Nested Transaction

- In nested transactions, top level transaction fork off children, which run in parallel to one another on different computers for improving the performance. Every forked child may also execute one or more subtransactions, or fork off its own children. Fig. 1.5.3 shows nested transaction.

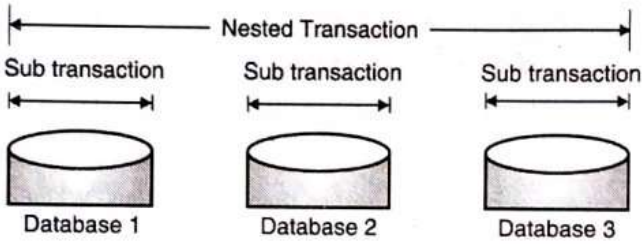


Fig. 1.5.3 : A nested Transaction

- Consider that, out of several subtransactions running in parallel one of the commits and submit the result to the parent transaction. If parent aborts after further computation and restores entire system to the previous state it had before the top-level transaction began.
- In this case the committed result of subtransaction must nevertheless be undone. Therefore the permanence referred to above applies only to top-level transactions. Any transaction or subtransaction basically works on a private copy of all data in the entire system.
- They manipulate the private copy only. If transaction or subtransaction aborts then private copy disappears. If it commits, its private copy replaces the parent's copy. Hence, if after commit of subtransaction a new subtransaction is started, then new one notice results created by the first one. Similarly, if higher-level transaction aborts then all its underlying subtransactions have to be aborted as well.
- Nested transactions present a natural means of distributing a transaction across multiple machines in distributed system. They exhibit logical splitting up of the work of the original transaction.

1.5.3 Enterprise Application Integration

- It became obvious to have some facilities to integrate applications independent from their databases. The reason behind this was the decoupling of applications from databases. There is requirement of straightway communication among application components instead of using the request/reply pattern that was supported by transaction processing systems.

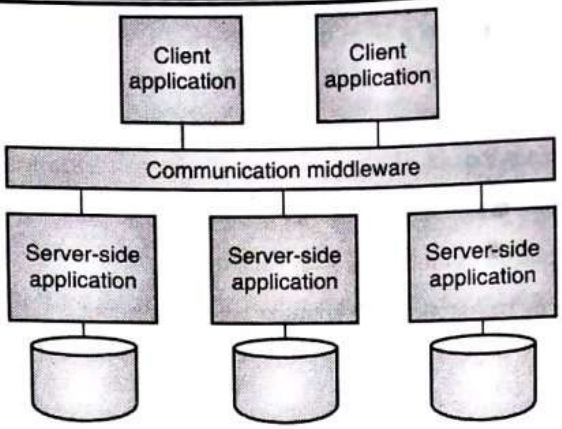


Fig. 1.5.4 : Middleware as a communication facilitator in enterprise application integration

- There are many examples of communication middleware. Using Remote procedure calls (RPC), an application component at client side can efficiently send a request to another application component at server side by making a local procedure call. This request at client side is packed as a message and sent to the called application. In the same way, the result will be packed as message and sent back to the calling application as the result of the procedure call.
- Like RPC, it is also possible to call the remote objects using remote method invocations (RMI). An RMI is basically is similar to RPC, except that it operates on objects rather than applications. While communicating using RPC or RMI both caller and callee have to be up and running. This is the drawback of it.
- In case of message-oriented middleware, applications just send messages to logical contact points, describe by means of a subject. Similarly, applications can demand for a specific type of message and communication middleware ensure that those messages are delivered to those applications.

1.5.4 Distributed Pervasive Systems

- The distributed system discussed above is stable and nodes are fixed and permanent in it. The distributed system in which mobile and embedded computing devices are present, instability is default behavior. These devices in the system called as distributed pervasive systems are small in size with battery-powered, mobile, and connected through wireless connection.
- An important characteristic of this system is absence of human administrative control. One possible best solution is owner configures their devices. These devices require discovering their environment automatically.
- In pervasive systems devices usually connect to the system with the intention of accessing and probably for providing the information. This calls for means to without difficulty read, store, manage, and share information. Considering irregular and altering connectivity of devices, the memory space where accessible information resides will most likely alter continually.

1.6 Distributed System Models

Basically distributed system models are categorized as :

- **Architectural Models :** These models concern about placement of different components of distributed system and relationship between these components.

- **Fundamental Models :** These models concern with formal description about properties that often appears and common in above architectural model.

1.6.1 Architectural Models

- These models of distributed system deals with placement of different components and with how these components are related to each other. Client server model is one of the examples of the architectural model. It is possible to modify client server model with following decisions. These are for examples :
  - Decision for partitioning the data and placing it at cooperating server machines.
  - Proxy client and servers for caching.
  - Client server model can also be modified by use of mobile and mobile agents.
  - As per requirement, adding and removal of mobile devices should support in suitable manner.
- The distributed system with decided components for design should meet current needs and future requirements. The designed distributed system should also supports for reliability. The system should be enough flexible to make changes, easy to manage, adaptable and should be cost effective.
- The architectural models considered here makes use of processes and objects.
- Three types of processes that are considered are (i) server (ii) client (iii) peer processes. Peer processes cooperate and interact with each other in symmetrical way to carry out the task.
- These processes can be placed at different machines in network in order to improve reliability and performance of the system. As a variation to client server model, it is possible to build dynamic system. Processes can move from one machine to other. Client can download code from server to run it locally. Communication traffic and access delays can be reduced by moving data from server to client machine.
- The design of some distributed system also allows adding or removing machines or mobile devices seamlessly. These devices can offer services to other or find out available services in the system.

1.6.1(A) Software Layers

- Software architecture defines software layers in single computer or it defines services that are provided or requested by processes running locally or on remote machines. This view can be expressed in terms of service layers.

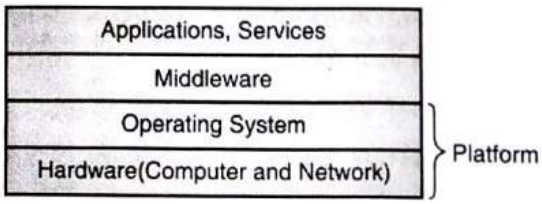


Fig. 1.6.1 : Service Layers in Distributed System

- Server process offers services to requesting client processes. Distributed Service can be placed on one or more servers which interact with each other and client processes by maintaining the consistent view of the service resources.
- In Fig. 1.6.1, lowest two layers are hardware and software layers which offer services to upper layers. These layers are platform for applications and distributed system. Middleware layer masks heterogeneity and offer communication and programming model to programmers.

- Middleware offers distribution transparency. It also provides services needed by application programs. Remote procedure call (RPC), Remote method invocation is the examples of middleware models.

1.6.1(B) System Architecture

- The key aspect in design of distributed system is placement of different system components such as applications, services servers and other processes on different computers in network. This decision affects performance, reliability and security of the resulting system.
- Following are the main types of architectural models.

Client Server Model

- This is basic and widely employed model. In this model server and client processes are on different computers. Client processes interacts with server processes to access the shared resources or to get services.
- In this model, requesting process is considered as client. Server process in turn may send request to other server process on different machine. For example web server may request local file server which manages files that contains web pages.

Services by Multiple Servers

- Several server processes, each running on separate computer may implement particular service. These server processes communicates with each other to provide the services to requesting client process. For example, client can access resource available on any particular web server. Here resources are partitioned and kept on different servers.
- Objects on which service is dependent can be distributed among many servers. In other case, objects can be replicated on many computers where server processes are running. The replication improves performance, availability and fault tolerance. Consistent copies are maintained at replicated serves. For example, web service can be mapped onto many servers having replicated database.

Proxy Servers and Caches

- Recently used data objects are maintained at cache. The newly received objects are added in cache by replacing existing objects there if required. Cached objects needed by client are first checked by caching service to confirm it as up-to-date copy. If not then up-to-date copy is accessed. These caches are collocated with other each client copies or may be fetched from proxy server shared by many clients.
- For example, Web browsers always caches recently used web pages, resources and maintain at client local file system. Before displaying these pages, browser uses HTTP request to check consistency of cached pages with server copies.
- Proxy servers can also maintain these shared caches to share among many clients. These proxy servers are maintained to reduce load on network and to improve performance. Proxy servers may also access remote web servers.

Peer Processes

- In this model, all processes are considered as equal and cooperate with each other to achieve a designated task. No process is client or server and code in these processes maintains the consistency in resources which are at application level. Application level activities are also synchronized by these processes whenever required

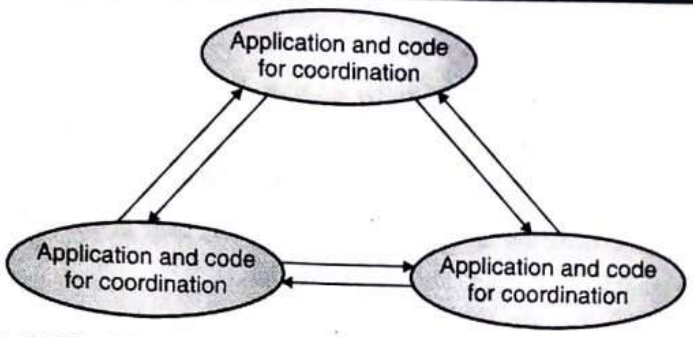


Fig. 1.6.2 : Distributed Computation with three peer processes

- In this computation, a delay incurred in communication with server processes is reduced. It leads to better interactive response among peer processes than server based architecture.

1.6.1(C) Variations in Basic Client Server Model

It is necessary to consider the following factors in design of client server model.

Mobile code and Mobile Agent

- Example of mobile code is downloading the applet code from web server by browser and running it locally. Sometimes server itself pushes up-to-date information to client. In this case client does not request server for the same information.
- Mobile agent is running program (code and data) which migrates from one machine to other in different network to perform a particular task.
- Mobile agent makes use of resources at machine where it is transferred. It may access database entries from database at this machine. Hence, communication cost can be reduced by avoiding the transfer of large size data from one machine to other machine.
- Mobile agents can be security threat at target machine where it is transferred. The environment receiving this mobile agent should check authentication of user of the mobile agent to access a particular resource. If mobile agent does not get required resource, it may not be able to complete the execution. Hence, mobile agents have limited applicability.

Network Computers

- Network computer downloads operating system and required application software from file server which is remote machine. In this case, although applications run on local machine but file server manages the files.
- The advantage of such configuration is that, user can work on any network machine as application data and code is stored at remote file server. The network computer need not have large memory capacity or processing power and hence cost can be reduced.
- Network computer uses resources at server. If disk is included then it stores minimum of software and used as cache to store recently accessed objects from remote file server. Cached objects at client are invalidated if gets updated at server.

The Clients

- Thin client is software layer which supports window-based user interface on computer where user is working while application is executing on remote server. This remote computer is powerful machine capable of executing large number of applications. It can be cluster computer or multiprocessor machine.
- Applications like CAD or image processing when executes at server, user at thin client notices delay as image and vector information needs to transfer from thin client to application.

Mobile Devices and Spontaneous Networking

- Laptops, PDAs, digital cameras, smart watches, mobile phones, washing machines and many such devices which can be connected with wireless networking can be part of distributed system.
- If these devices are integrated in distributed system then it supports for mobile computing. In this environment, user with mobile devices can access local or remote services.
- Spontaneous networking integrates these mobile devices in network. Spontaneous networking encompasses applications that involve connection of both mobile and non-mobile devices in ore informal manner.
- Spontaneous networking offers easy connection to local network and easy integration with local services. A communication link is provided to devices and after connection established devices access the required services available in network.
- Spontaneous networking design raises issues such as convenient connection and integration if machine or device moves to other network. Addressing and routing algorithms differs in other networks and users may get limited connectivity while traveling. Security and privacy issues need to be addressed as user gets connected in new network.

1.6.1(D) Interfaces and Objects

- Server processes can be implemented in object oriented languages such as C++ and Java. These server processes or peer processes encapsulates the objects which can be accessed by remote processes. The object reference can be passed to remote process so that methods implemented in objects can be invoked by remote processes with remote invocation.
- CORBA and Java supports such remote method invocation (RMI). Set of functions for invocation in server or peer process is specified by one or more interface definitions. The client server model can be static or more dynamic depending on decision about how to distribute responsibilities between processes and between machines in the network.
- In static architecture pre-allocation of responsibilities is done statically. For example, file server for file management. Objects can be dynamically instantiated whenever required for invocation.

1.6.1(E) Design Requirements for Distributed Architectures

Performance Issues

Limited communication and processing capabilities of computers and networks is considered to discuss performance issues. These are as follows :

Responsiveness

- Users of interactive applications expect fast and consistent response. To access remote service, fast response is not dependent only on load and performance of server, but it can be due to delay in software components. These software components are client and server operating systems, middleware and code that implements remote service.
- There should be less software layers and less amount of data transfer between client and server to improve response time. Caching of data at client side also gives faster response. Accessing remote data may result in variation of delay. For text data short delay and for graphical images longer delay will be incurred.

Throughput

- It is computational work carried out per unit time. Throughput is affected by processing speed at client and server side and data transfer rate between client and server.
- Data transfer between various software layers in both the sender and receiving machine also affects the overall throughput. Network should also support for faster data transfer rate to improve throughput.

Balancing Loads

- Instead of running all the services, applications, processes components at server, it is better to shift some execution at client side. For example, web servers load is reduced by running applets at client side.
- Information can be replicated on many web servers to distribute the requests from many clients. Partially completed jobs can be migrated to other machine for execution to balance the load.

Quality of Service

- Clients and users experiences quality of service due to system properties such as reliability, security and performance. System configuration may change or resources may not available. This also affects the quality of service. How users or applications adapt to changing system configuration is important.
- Reliability and security aspects of system are important design issue. System should supports for failure free operations and failure should not affect the system performance. System should also provide security to get quality of service.
- Performance factors which are responsiveness and throughput is also important that affect quality of service. Faster response and better throughput improves quality of service.
- Stream oriented communication involves time critical data transfer and processing. For example, fetching movie video from web server. To guarantee quality of service, network resources should be available along the route of data transfer. These resources involve computational capacity, bandwidth, buffer space etc.

Caching and Replication

- Data replication and caching improves performance of the distributed system. Replication is placing same data or information on several machines. Response from web server is cached by client and proxy server. The cache consistency protocol configured with web browser ensures to get up-to-date copy of resource cached by client.

- In order to take care of performance, availability and operations that are disconnected, this up-to-date condition can be relaxed. Client and proxy server sends request to original web server to validate cached copy. In case of failed response, server responds with up-to-date copy of cached data.
- Web server keeps record of browsers and proxies which have accessed the resource. The expiry time for this cached resource is set by server. This estimated expiry time is based on last updated time of accessed resource. Response from web server always contains expiry time of resource and current time.
- This cached response enables the browsers and proxies to know if it is likely to become stale for future request. Cached response age gets compared with expiry time. Age of cached response involves sum of time of response from server that has been cached and server time. This agreement of time is independent of the clock time at client, proxy or web server.

Dependability

- The users of the designed system are dependent on correctness, security and fault tolerance of system. Security and fault tolerance is need and these parameters have impact on the architectural design of the system.
- Although fault occurs in hardware, software or network, the applications which are dependent on the system should carry out the designated task correctly. Multiple copies of the resources should be provided so that system and application software can be reconfigured to complete its task.
- There is practical limitation to make available redundant resources and hence limitations to achieve expected degree of fault tolerance. Redundancy can be possible if at architectural level, multiple machines are available to run the component process of the system. In the same way, multiple communication paths are also required to exchange multiple messages between processes.
- Data can be replicated at several locations so that data access can be possible from machine which is currently running and with no any types of fault in it. Replication of data also incurs the cost of maintaining consistency between all copies.
- Architectural impact due to need of security enforces keeping of sensitive data and other resources on machine which is secured against many attacks and vulnerabilities.

1.6.2 Fundamental Models

- Fundamental models are based on fundamental properties that permit the users to be more specific about the characteristics, risk of failure and security.
- These models mainly consider primary entities of the system, their interaction and characteristics which impacts the individual or cooperative behavior of the system. In fundamental model, the main aspects focused are interaction, failure and security.

1.6.2(A) Interaction Model

- Server processes may interact with each other to provide services to client as per their requests. On the other hand peer processes may communicate with each other to achieve a common goal.

- In this scenario, many messages gets exchanged between these processes and state of these each interacting processes is private which cannot be accessed or updated by other process. Following factors affects interacting processes.

Communication Performance

- Latency, bandwidth and delay affect communication performance. Latency is time elapsed between start of transmission of message by sender process and when receiver process begins the receiving of the same message.
- Delay incurred in accessing the network that increases when network is heavily loaded due to traffic. Delay can be due to load at operating system at sender and receiver machines.
- Bandwidth of network decides transfer of information in given time. If more connections or channels use such network then bandwidth gets divided. Jitter is variation in consecutive packet or message delivery time. It is more related to multimedia data transfer.

Machine Clocks and Timing Events

- Different machines in the network have internal clock which is used by processes running on these machines to obtain current time. Practically, these clocks drift with respect to perfect time. Some machines clock can be leading or lagging with respect to perfect or standard time. Hence, processes running on different machines read different clock values as this machine's clock time are not same.
- The drift rate of different machines clock is not similar. The drift rate is defined as relative amount the machine clock differs with respect to perfect clock. Hence, although all machines clock are set at same time initially, in future these clocks would differ in time.
- Several algorithms are used to synchronize the clocks of computer in the network.

Two variants of interaction models

It is difficult to set time limit for execution of processes, delivery of messages to processes running on other machine and drift rate of clocks of different machines. Following are two examples of interaction model. One model requires strong assumption about time and other requires no assumption about time.

Synchronous Distributed Systems

- In synchronous distributed system, following bounds are defined :
- Lower and upper bounds of each process for their execution of each step are known in advance.
- Message will be received by each process from other process running on different machine in bounded time.
- The machine on which process is running have known bound about its local clocks drift rate with respect to real time clock.
- It is possible to model such synchronous distributed system. Practically, the above defined bounds should be guaranteed. It is necessary to give guarantee about required processor cycles and network capacity to use resources to complete tasks by processes within defines time bounds.

Asynchronous Distributed Systems

Internet is example of asynchronous distributed system. This system does not consider timing bounds. For example, if user is experiencing delay in receiving response from web server then web browser allows user to carry out other tasks. In this type of distributed systems, there are no bounds on :

- **Speed of process execution :** Processes which are communicating with each other may take different times to finish the particular step. For example, requesting process step may execute in nanosecond and responding process step on other machine may take longer time (may be second, minutes and more than that).
- **Delays in message transmission :** One message from process A may take time in picoseconds to deliver to process B whereas other message from the same process may take hours.
- **Drift rate of clock :** It can be arbitrary for different clocks of different machines.
- Practically, asynchronous distributed system is very often as it is necessary to share processors and communication channels in network to accomplish the designated task. Hence, in asynchronous distributed system it is possible to implement the bounds defined above for synchronous distributed system. For example, multimedia data stream can be delivered within bounded time in asynchronous distributed system.

Ordering of Events

- Ordering of events between communicating processes is very important to achieve consistency at different sites. For example, two servers in different cities A and B maintains bank database. For example, customer accounts balance initially is Rs 1000. If customer deposited Rs 100 in city A and at the same time branch manager gives 1% interest on account balance in city B.
- These are two events related to same copy of database that is present on two different servers. Lets us say these events as event A (updating account balance by adding Rs 100) and event B (updating account balance by adding 1% interest). If these events executes in different order at these databases then inconsistency in account balances would occur.
- In city A, if event A is executed first and then suppose event B is executed. In this case account balance in city A will be updated as: 1000 + 100 = 1100 and then adding Rs 11 interest = **Rs 1111**. In city B, if event B is executed first and then suppose event A is executed. In this case, account balance in city B will be updated as : 1000 + 10 = 1010 and then adding Rs 100 = Rs **1110.**
- These Event A and B messages arrive at both sites A and B in order explained above due to communication delay in network. If clocks of all machines in network show same time then ordering of messages can be done as per timestamp when message was sent. Practically, it is not possible. Lamport proposed notion of logical time to arrange the messages at different sites in same order for execution. The logical clock concept will be explained in chapter 3.

1.6.2(B) Failure Model

In distributed system, communication channel, processes, machines can fail during the course of execution. Failure model suggest the ways in which failures in different components of the system may occurs and to provide the understanding of this failure.

Omission Failures

The omission failure refers to the faults that occur due to failure of process or communication channel. Because of this failure, process or communication channel fails to carry out the action or task that it is supposed to do.

Process Omission Failures

- The main omission failure of process is when it crashes and never executes its further action or program step. In this case, process completely stops. When any process does not responds to requesting process repeatedly, the requesting process detects or concludes this crash.
- The detection of crash in above manner relies on use of timeouts. In asynchronous system, timeout can point towards only that process is not responding. The process may have crashed, may be executing slowly or messages have yet not delivered to the system.
- If other process surely detects the crash of process then this process crash is called as fail-stop. This fail-stop behaviour can be formed in synchronous system when processes uses time out to know when other process fail to respond and delivery of messages are guaranteed.

Communication Omission Failures

- Sending process P executes *send* primitive and puts message in its outgoing buffer. Communication channel transport this message to receiving process Q's incoming buffer. Process Q then executes *receive* primitive to take this message from its incoming buffer. It then deliver the message.
- If communication channel is not able to transport message from process P's outgoing buffer to process Q's its incoming buffer then it produces omission failure. The message may be dropped due to non-availability of buffer space at receiving side, no buffer space at intermediate machine or network error detected by checksum calculation with data in message.
- Send-omission failures refer to loss of messages between sending process and its outgoing buffer. Receive- omission failures refer to loss of messages between incoming buffer and the receiving process.

Arbitrary Failures

- In this type of failure process or communication channel behaves arbitrarily. In this failure, the responding process may return wrong values or it may set wrong value in data item. Process may arbitrarily omits intentional processing step or carry out unintentional processing step.
- Arbitrary failure also occur with respect to communication channels. The examples of these failures are : message content may change, repeated delivery of the same messages or non-existent messages may be delivered. These failures occur rarely and can be recognized by communication software.

Timing Failures

- In synchronous distributed system, limits are set on process execution time, message delivery time and clock drift rate. Hence, timing failures are applicable to this system.
- In timing failure, clock failure affects process as its local clock may drift from perfect time or may exceeds bound on its rate.

- Performance failure affects process if it exceeds the defined bounds on the interval between two steps.
- Performance failure also affects communication channels if transmission of message take longer time than defined bound.

Masking Failures

- Distributed system is collection of many components and components are constructed from collection of other components. Reliable services can be constructed from the components which exhibit failures.
- For example, suppose data is replicated on several servers. In this case, if one server fails then other servers would provide the service. Service mask failure either by hiding it or by transforming it in more acceptable type of failure.

1.6.2(C) Security Model

- The architectural model discussed offers the basis for security model. The distributed system can be secured by providing security to processes and to the communication channels through which these processes communicates. Security also can be achieved by protecting the objects which these processes encapsulate against unauthorized access.
- **Protecting Objects :** Server keeps objects to which invocations come from clients at different machines. Invocations may be from process or user. These objects at server hold users private data or shared data such as web pages. It is necessary to provide access right to users with each invocation. Server should verify this authority of invocation and access rights for it.
- **Securing Processes and their Interactions :** Processes communicates with each other by sending the messages to each other. These messages are exposed against various attacks while under travel. Peer processes and server processes expose their interfaces through which other processes send them invocations. Integrity should be maintained for applications which handle financial transaction or confidential data.
- **Adversary :** Threat may come from legitimate machine in network or from machine which is connected in unauthorized way. Attacker may be capable of sending the messages to any process or may read or copy messages while in transmission. Incoming request to process may be from unauthorized source. This source address can be forged by attacker. Server also gets invocation requests from many clients. Server cannot accept or reject these requests without reliable knowledge of senders.
- **Uses of Security Models :** It is not true that, distributed system can be secured by securing communication channels and by applying access rights only. Encryption or access right techniques incur high processing and management cost. This analysis can help to develop threat model by considering all attacks, its sources and by minimizing the cost above.

1.7 Hardware Concepts

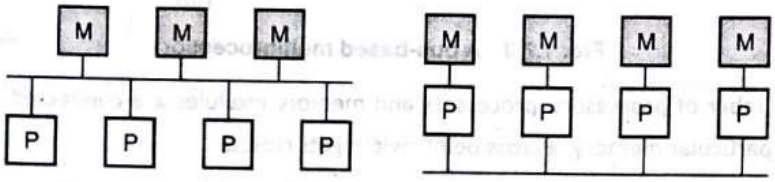
Hardware concepts illustrate the organization of hardware, their interconnection and the manner in which it communicates with each other. Multiple CPUs exist in distributed system.

The machines are basically divided in two groups

- **Multiprocessors :** Processors share the memory.
- **Multicomputers :** Each processor has its own private memory.

Both multiprocessors and multicomputers further divided in two categories on the basis of architecture of interconnection network. The basic organization is shown in Figs. 1.7.1 and 1.7.2.

- **Bus-based :** Machines are connected by single cable or processors are connected by bus and share the memory by communicating over the bus.
- **Switch-based :** Machines are connected with different wiring patterns and messages are routed along outgoing line by switching the switch.

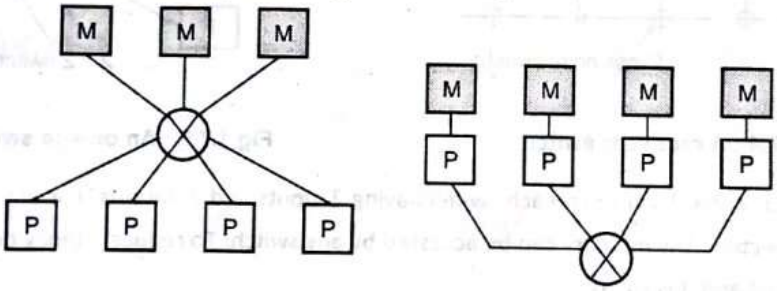


Bus-based shared memory multiprocessor Processors share the memory

Bus-based multicomputer. Each processor has its private memory

Fig. 1.7.1

- Multicomputers can be homogeneous or heterogeneous. Homogeneous multicomputers use same technology for interconnection network and all processors are similar, accessing the same amount of private memory.
- Heterogeneous multicomputers have different architecture for different machines. Machines are connected by different types of networks using different technology.



Processors share the memory and interconnected through switch

Processors have private memory and interconnected through switch

Fig. 1.7.2

1.7.1 Multiprocessors

- Multiple CPUs and memory modules are connected through bus. Single memory is available. CPU1 writes memory and CPU2 reads written value. CPU2 gets the same value which was written by CPU1. Memory having such property is said to be coherent.
- The disadvantage of the scheme is that, if few numbers of processors are connected then bus becomes overloaded and performance degrades. To overcome the problem, a high cache memory is placed between CPU and bus. If referenced word is in cache then it is accessed by CPU and bus is avoided. Typical size of cache is 512 KB to 1 MB.

- Again memory can be incoherent if updated word in one cache is not propagated to other copies of the word presen in different cache memory : otherwise other copies will have old values. A bus-based multiprocessor provides les scalability.

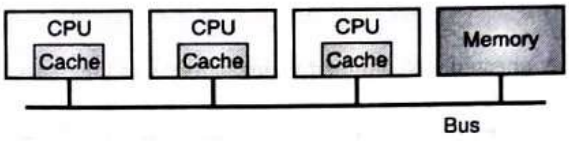


Fig. 1.7.3 : A bus-based multiprocessor

- To allow for more number of processors, processors and memory modules are connected by crossbar-switch. Whe CPU wants to access particular memory, a cross point switch gets closed.
- Several CPU can access different memory simultaneously but for two CPUs to access same memory at the same time one has to wait. It is shown in Fig. 1.7.4 Here for n CPUs and n memories, n^2 cross point switches are required.

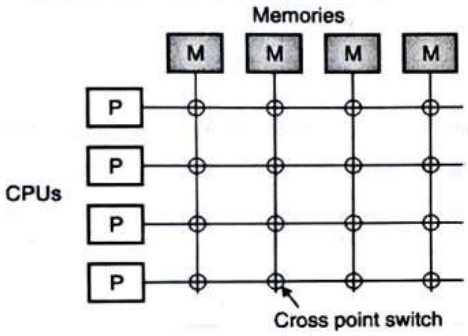


Fig. 1.7.4 : A crossbar switch

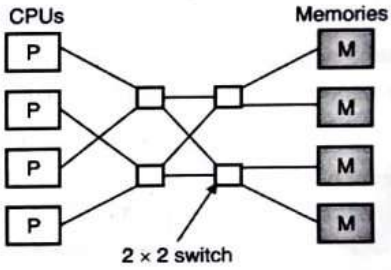
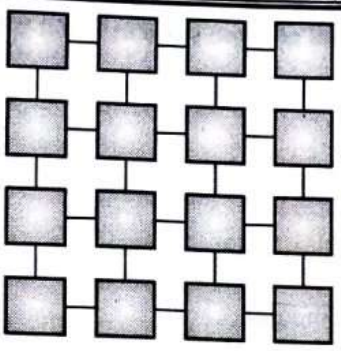


Fig.1.7.5 : An omega switching network

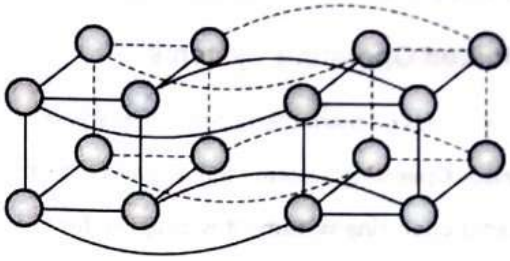
- Omega network contains 2 X 2 switches, each switch having 2 inputs and 2 outputs. For any input any one out of two output line can be selected. Any memory can be accessed by any switch. To reduce latency between CPU and memory fast switching is needed and it is costly.
- Non-uniform memory access architecture allows the CPU to access its local memory fast and other CPU's memory is slowly accessed.

1.7.2 Homogeneous Multicomputer Systems

- Nodes are connected through high speed interconnection network. Processors communicate through interconnection network; therefore traffic is low with compare to traffic between processor and memory.
- If nodes are connected through interconnection networks like fast Ethernet then it is bus-based multicomputer. It has limited scalability and performance is low if few number of processors (25-100 nodes) added.
- Instead of broadcasting like in bus based multicomputers, routing of messages can be done through interconnection network. A grid is shown in Fig. 1.7.6 (a) and suitable for graph theory related problems. A 4-dimentional hypercube is shown in Fig. 1.7.6 (b). Vertices represent CPU and edges represent connection between CPUs.



(a) Grid



(b) Hypercube

Fig. 1.7.6

- Massively parallel processors (MPPs) and Cluster of workstations are the examples of the switched multicomputers.

1.7.3 Heterogeneous Multicomputer Systems

- Today most of the distributed systems are built on the top of multicomputers having different processors, memory size etc. Interconnection network also have a different technology. These are called as heterogeneous multicomputers. In large scale heterogeneous multicomputer environment, services to application and its performance varies at different locations.
- Absence of global system view, intrinsic heterogeneity, scaling are the factors due to which there is a requirement of sophisticated software to built the distributed applications for heterogeneous multicomputers.

1.8 Software Concepts

Distributed systems and traditional operating systems provides the similar services such as :

- Both play the role of resource **managers** for the underlying hardware.
- Both permit the sharing of resources like CPUs, memories, peripheral devices, the network, and data of all kinds among multiple users and applications.
- Distributed systems try to hide the details and heterogeneous nature of the underlying hardware by offering a virtual machine on which applications can be executed without trouble.
- In a network multiple machines may have different operating system installed on it. Operating systems for distributed computers are categorized as :
- **Tightly coupled systems :** It keeps a single, global view of the resources it manages. Such tightly coupled operating system is called as distributed operating system (DOS). It is useful for the management of multiprocessors and homogeneous multicomputer. DOS hides details of underlying hardware. This hardware can be shared by many processes and details remains hidden.
- **Loosely-coupled systems :** In a set of computers, each has its own OS and there is coordination between operating systems to make their own services and resources available to the others. This loosely coupled OS is called as network operating system (NOS) and is used for heterogeneous multicomputer systems.

- Apart from above operating systems **middleware** is used to provide general purpose services and it is installed on the top of NOS. It offers distribution transparency.

1.8.1 Distributed Operating Systems

There are two types of distributed operating systems.

- **Multiprocessor Operating System :** It manages resources of multiprocessor.
- **Multicomputer operating system :** It is designed for homogeneous multicomputer.
- Distributed operating system is similar in functions to the traditional uniprocessor operating system. DOS handles the multiple processors.

Uniprocessor Operating Systems

- These are traditional operating systems and developed for single CPU. It permits the user and applications to share the different resources available in system. It allows the different applications to use same hardware in isolated manner. Simultaneously several applications executing on the same machine gets their required resources.
- Operating system protects data of one application from the other application if both are simultaneously executing. Applications should use only facilities provided by OS. To send the messages, applications should use communication primitives offered by OS.
- Operating system takes care of the way in which hardware resources are used and shared. For this purpose CPU supports for two modes of operation, kernel mode and user mode.
- In **kernel mode**, execution of all instructions is allowed to be carried out, and the entire memory and set of all registers is accessible throughout execution. On the contrary, in **user mode**, there is a restriction on memory and register access. When CPU executes operating system code, it switches to kernel mode. This switching from user mode to kernel mode occurs through system calls that are implemented by the operating system.
- If virtually all operating system code is executes in kernel mode then operating system is said to have monolithic architecture and it runs in single address space. The disadvantage of monolithic design approach is lack of flexibility. In this design, it is difficult to replace the components.
- Monolithic operating systems are not a good scheme from point of view of openness, software engineering, reliability, or maintainability. If the operating system is organized into two parts, it can provide more flexibility. In the first part, set of modules for managing the hardware is kept which can uniformly well be executed in user mode.
- For example, memory management module keeps track allocated and free space to the processes. It is required to execute in kernel mode at the time when registers of the Memory Management Unit (MMU) are set.
- A small **microkernel** contains only code that must execute in kernel mode. It is the second part of the operating system. Actually, a microkernel require only contain the code for, context switching, setting device registers, manipulating the MMU, and capturing hardware interrupts.
- Microkernel also contains the code to pass system calls to calls on the proper user level operating system modules, and to return their results. Following is the organization of operating system with this approach.

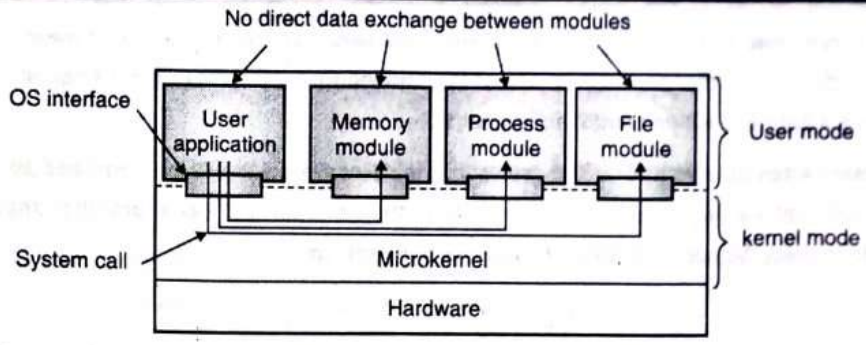


Fig. 1.8.1 : Separating applications from operating system code through a microkernel

Multiprocessor Operating Systems

- Unlike the uniprocessor operating systems, multiprocessor operating systems offers support for multiple processors having access to a shared memory. In this case, all data structures required by the operating system to deal with the hardware, together with the multiple CPUs, are placed into shared memory. Multiple processors can access these data. Hence protection against simultaneous access is needed to promise consistency.
- Modern operating systems are designed with the intension of handling multiple processors. The main aim of multiprocessor operating systems is to achieve high performance by means of multiple CPUs. A key goal is hide the presence of number of CPUs from the application.
- Such transparency can be attained and is relatively straightforward as different parts of applications communicates by using same primitives as those in multitasking uniprocessor operating systems. All the communication is made by manipulating data at shared memory locations, and it is required protect that data against concurrent access. This protection is achieved through synchronization primitives : semaphores and monitors.

Semaphores

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations : wait and signal. These operations were firstly termed P (for wait) and V (for signal).
- A semaphore S is integer variable whose value can be accessed and changed only by two operations wait (P or sleep or down) and signal (V or wakeup or up). Wait and signal are atomic operations.
- Binary semaphores do not assume all the integer values. It assumes only two values 0 and 1. On the contrary, counting semaphores (general semaphores) can assume only non-negative values.
- The wait operation on semaphores S, written as wait(S) or P(S), operates as follows :

```
wait(S) : IF  S > 0
          THEN S : = S – 1
          ELSE  (wait on S)
```

- The signal operation on semaphore S, written as signal(S) or V(S), operates as follows :

```
signal(S) : IF (one or more process are waiting on S)
            THEN (let one of these processes proceed)
            ELSE  S : = S +1
```

- The two operations wait and signal are done as single indivisible atomic operation. It means, once a semaphore operation has initiated, no other process can access the semaphore until operation has finished. Mutual exclusion on the semaphore S is enforced within **wait(S)** and **signal(S)**.
- If many processes attempt a wait(S) at the same time, only one process will be permitted to proceed. The other processes will be kept waiting in queue. The implementation of wait and signal promises that processes will not undergo indefinite delay. Semaphores solve the lost-wakeup problem.

Monitors

- If semaphores are used incorrectly, it can produce timing errors that are hard to detect. These errors occur only if some particular execution sequences results and these sequences do not always occur.
- In order to handle such errors, researchers have developed high-level language constructs called as monitor.
- A monitor is a set of procedures, variables, and data structures that are all grouped together in a particular type of module or package. Processes may call the procedures in a monitor if required, but direct access to the monitor's internal data structures from procedures declared outside the monitor is restricted to the processes.

Following is the example of the monitor.

```
monitor example
integer i ;
condition c ;
procedure producer ( );
end ;
procedure consumer ( );
end ;
end monitor ;
```

- Monitors can achieve the mutual exclusion: only one process can be active in a monitor at a time.
- As monitors are a programming language construct, the compiler manages the calls to monitor procedures differently from other procedure calls.
- Normally, when a process calls a monitor procedure, if any other process is currently executing within the monitor, it gets checked.
- If so, the calling process will be blocked until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

The characteristics of a monitor are the following

- Only the monitor's procedures can access the local data variables. External procedures cannot access it.
- A process enters the monitor by calling one of its procedures.
- Only one process may be running in the monitor at a time; any other process that has called the monitor is blocked, waiting for the monitor to become available.

- Synchronization is supported by monitor with the use of **condition variables** that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions :
- **cwait (c) :** the calling process's execution gets suspended on condition c. The monitor is now accessible for use by another process.
- **csignal (c) :** blocked process after a cwait (on the same condition) resumes its execution. If there are many such processes, choose one of them; if there is no such process, do nothing.
- Monitor wait and signal operations are dissimilar from those for the semaphore. If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.
- When one process is executing in monitor, processes that trying to enter the monitor join a queue of processes blocked waiting for monitor availability.
- Once a process is in the monitor, it may temporarily block itself on condition x by issuing cwait (x); it is then placed in a queue of processes waiting to reenter the monitor when the condition changes, and resume execution at the point in its program following the cwait (x) call.
- If a process that is executing in the monitor detects a change in condition variable x, it issues csignal (x), which alerts the corresponding condition queue that the condition has changed.
- A producer can add characters to the buffer only by means of the procedure append inside the monitor; the producer does not have direct access to buffer.
- The procedure first-checks the condition not full to determine if there is space available in the buffer. If not, the process executing the monitor is blocked on that condition.

Multicomputer Operating Systems

- In multicomputer operating systems data structures for systemwide resource management cannot simply shared by keeping them in physically shared memory. In its place, the only way of communication is through **message passing**. Following is the organization of multicomputer operating systems shown in Fig 18.2.

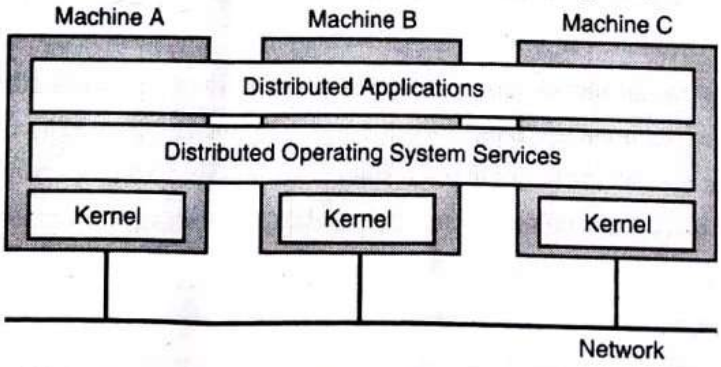


Fig. 1.8.2 : General structure of a multicomputer operating system

- Kernel on each machine manages local resources, for example memory, the local CPU, a local disk and others. Each machine contains separate module for sending and receiving messages to and from other machines.

- On the top of each local kernel there is a common software layer that implements the operating system as a virtual machine supporting parallel and concurrent execution of various tasks.
- This software layer offers a complete *software implementation* of shared memory. Further facilities usually implemented in this layer are, such as, assignment of task to processor, masking hardware failures, providing transparent storage, and general interprocess communication.
- Some multicomputer operating systems offers only message-passing facilities to applications and do not offer shared memory implementation. But different system can have different semantics of message-passing primitives. Their dissimilarities can be explained by taking into consideration whether or not messages are buffered. When should be blocking of sending or receiving process is also need to be considered as well.
- Buffering of the messages can be done at the sender's side or at the receiver's side. There are four synchronization points at which a sender or receiver can possibly block. At the sender's side, sender is blocked if buffer is full. If sender buffer is not present then three other points to block the sender are :
  - The message has been sent by sender.
  - The message has arrived at the receiver side.
  - The message has been delivered to the receiver application.
- Another issue that is significant to know message-passing semantics is whether or not communication is reliable. In reliable communication, a assurance of receiving the message by receiver is given to the sender.

Distributed Shared Memory Systems (DSMs)

- Programming for multicomputers is complex with compare to multiprocessors. As only message passing is available with multicomputers, programming becomes difficult with multicomputers. On the other hand, multiprocessor uses semaphores and monitors to access the shared data. So programming with multiprocessors is simple. In multicomputer case buffering, blocking, and reliable communication needs to be consider as well which leads to complex task of programming.
- Implementing the shared memory on multicomputers offers a virtual shared memory machine, running on a multicomputer. This shared memory model can be used to write the applications on multicomputers. The role of multicomputer operating system is important in this case.
- A large virtual address space can be created by using virtual memory of each individual node. With this approach a page based **distributed shared memory (DSM)** is realized. In DSM system, the address space is broken into pages of size 4 KB or 8 KB and kept over all the nodes in the system. If a processor reference fails to an address which is locally unavailable, a trap takes place, and the operating system fetches the page having the referenced address and starts again the faulting instruction, which now completes successfully.
- Following Fig. 1.8.3 shows an address space divided in 16 pages and four processors are available. It is just like normal paging. Here RAM of other machine is being used as the backing store instead of the local disk.

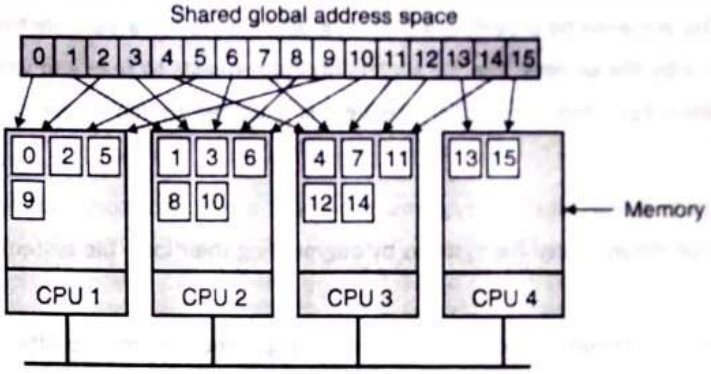


Fig. 1.8.3 : Pages are spread over 4 machines

- As shown in figure, if processor 4 references the instruction or data belonging to the pages 13 and 15 then it is done locally. If references to instructions and data are which are in the pages placed on other machines then trap to the operating system will occur.
- All read only pages can be replicated so that references will be done locally and performance can be improved. If read-write pages are replicated then if one of the copies gets modified other copies should also reflect the same changes. Write invalidate can be used to perform write on the page. Before performing write other copies are invalidated.
- If size of the pages kept large then it can reduce the total number of transfers when large section of contiguous data needs to be accessed. Conversely, if a page comprises data needed by two independent processes on different processors, the operating system may need to repetitively transfer the page between those two processors. This is called as **false sharing**.

1.8.2 Network Operating Systems

- Network operating systems assumes underlying hardware as heterogeneous. Whereas DOS assumes underlying hardware as homogeneous. In heterogeneous environment, machines and operating systems installed on them may be different. All these machines are connected to each other in computer network.
- NOS permit users to use services available on a particular machine in the network. Remote login service provided by NOS allows the user to log in remote machine from his/her terminal. Using command for remote copy user can copy the file from one machine to other.

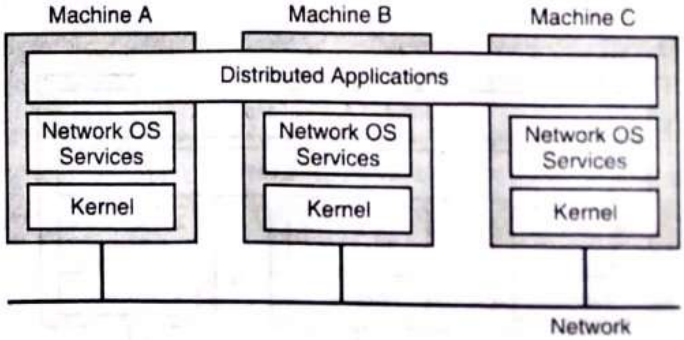


Fig. 1.8.4 : General structure of a network operating system

- Information sharing can be achieved by providing a shared, global file system accessible from all the workstations. The file system is implemented by **file servers**. The file servers accept requests to read and write files from user programs running on **client machines.** Each request of client request is executed by file server, and the reply is sent back to client.
- File servers usually supports hierarchical file systems, each with a root directory including subdirectories and files. Workstations can import or mount these file systems by augmenting their local file systems with those located on the servers.
- Distributed operating system attempts to achieve complete transparency in order to offer a single system view to the user. Whereas achieving full transparency with network operating system is not possible. As we have seen for remote login, user has to explicitly log into remote machine. In remote copy user knows the machine to which he/she is copying the files. The main advantage of network operating system is that it provides scalability.

1.9 Middleware

- Both DOS and NOS do not fulfil the criteria required for distributed system. A distributed operating system cannot manage a set of *independent* computers, while a network operating system does not provide transparency. A distributed system consisting of advantages of both DOS and NOS : the scalability and openness of network operating systems (NOS) and the transparency and related ease of use of distributed operating systems (DOS) is possible to realize by using middleware.
- A middleware layer can be added on the top of network operating systems which hide the heterogeneity of the collection of underlying platforms but as well get the better distribution transparency. The most of the modern distributed systems are build by means of such an additional layer of what is called **middleware**.

1.9.1 Positioning Middleware

- It is not possible to achieve distribution transparency as many distributed applications make direct use of the programming interface provided by network operating systems. Applications always make use of interfaces to the local file system as well.
- If additional layer of software is placed between applications and the network operating system then distribution transparency can be achieved and higher level of abstraction is provided. This layer is called **middleware** as shown in Fig. 1.9.1.

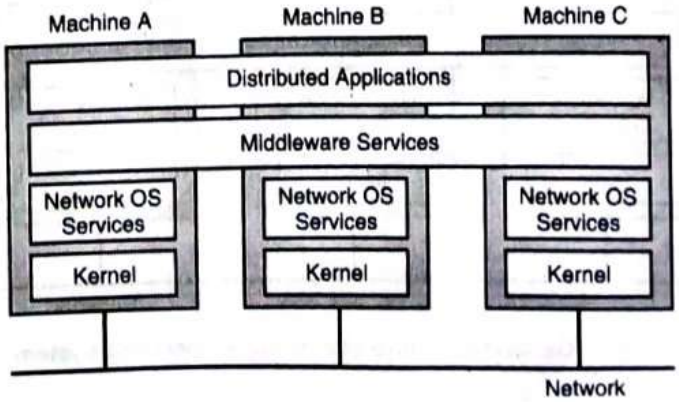


Fig. 1.9.1 : organization of distributed system as middleware

- In this organization local resource management is handled by NOS. It also provides simple communication means to connect to other computers in the network. A major goal is to hide heterogeneity of the underlying platforms from applications. As a result, many middleware systems offer an essentially complete set of services.

1.10 Models of Middleware

- In order to ease the development and integration of distributed applications most middleware is uses some model to express distribution and communication. A relatively simple model is that of treating everything as a file.
- The approach of treating everything as a file is introduced in UNIX and also followed in plan 9. Here, all devices such as keyboard, mouse, disk, network interface, etc, are treated as files. In essence, whether a file is local or remote makes no difference. For reading or writing bytes in file, an application first opens the file and then perform read or write operation. Once the operation is done it closes file again. As files can be shared by many processes, communication reduces to just accessing the same file.
- Another model of middleware is based on **distributed file systems**. This model supports distribution transparency only for files that only stores data. This model became popular as it is reasonably scalable.
- In **Remote Procedure Calls (RPCs)** based middleware model, a client side process calls a procedure implemented on a remote machine. In this call, parameters are transparently sent to the remote machine (server) where the procedure actually executed. The result of execution then server sent back to the caller. Although called procedure is executed on remotely, it appears as call was executed locally. In this case the communication with remote process remains hidden from calling process.
- Similar to calling the remote procedure in RPC, it is also possible to invoke objects residing on remote computers in a transparent manner. Many middleware systems offer a concept of **distributed objects**. The main idea behind distributed objects is that each object implements an interface that hides all the internal details of the object from its users. An interface contains the methods that the object implements. The process notices of an object are its interface.
- In the implementation of distributed objects, object resides on single machine and its interface is kept on several other machines. The available interface of the object on invoking process's machine converts its method invocation into a message that is sent to the object. After receiving method invocation message of process, object executes the invoked method and sends back the result. The interface implementation then converts the reply message into a return value, and submits to the invoking process. Similar to RPC, the invoking process remains totally unaware of the network communication.
- The World Wide Web became successful due to the very simple but very much effective model of **distributed documents**. In web model, information is organized in the form of documents. Each of the document is kept on a machine transparently sited anywhere in the wide area network.
- Links in the documents refer to other documents. By using link referring to particular document, that document is obtained from its location and gets displayed on the user's machine. Apart from textual documents, web also supports for audio, video and interactive graphic-based documents.

1.11 Services Offered by Middleware

- All middleware implements access transparency. For this, they provide high-level **communication facilities** to hide the low-level message transmitting through computer networks. How communication is supported depends very much on the model of distribution the middleware offers to users and applications.

- **Naming** service is offered by almost all middleware. Due to name services offered by middleware it is possible to share and search for the entities. Naming service becomes complex to deal with if scalability is considered. For efficient searching of a name in a large-scale system, the place of the entity that is named must be assumed to be fixed. This main difficulty which is needed to be handled. In World Wide Web (WWW), URL is used to refer to the document. Server on which this document is stored, its name is present in URL. Therefore, if the document is migrated to another server, its URL fails to search the document.
- **Persistence** service is offered for storage purpose. Persistence service is provided through a distributed file system in simple way, but many advanced middleware have integrated databases into their systems. If not, it provides facilities for applications to connect to databases.
- Many middleware offers facility for **distributed transactions** if data storage is important. Transactions permit multiple read and write operations to take place atomically. Atomicity is the property where either transaction commits so all its write operations are actually performed, or it fails, leaving all referenced data unchanged. Distributed transactions operate on data which is distributes across multiple machines.
- The important service provided by middleware is **security.** Instead of depending on the underlying local operating systems to sufficiently support security for the entire network, security has to be partly implemented additionally in the middleware layer itself.

Middleware and Openness

- Modern distributed systems are built as middleware for different operating systems. Due to this, applications developed for a specific distributed system become operating system independent and more dependent on specific middleware. The difficulty arises due to less openness of middleware.
- Actual open distributed system is specified through interfaces that are complete. Completeness states that the details required for implementing the system, has surely been specified. If interfaces are not complete then system developers must add their own interfaces. As a result, situation arises in which two middleware systems developed by different teams follow the same standard, but applications developed for one system cannot ported to the other without trouble.
- Due to incompleteness, although two different implementations implement precisely the same set of interfaces but different underlying protocols, they cannot interoperate.

Comparison between DOS, NOS and Middleware

Sr. No.	Distributed OS (DOS)	Network OS(NOS)	Middleware based DS
1.	1. Degree of transparency is very high for multiprocessor OS. 2. Degree of transparency is high for multicomputer OS.	Degree of transparency is low.	Degree of transparency is high
2.	Same operating system is present on all nodes in both cases (Multiprocessor OS and Multicomputer OS).	Operating system on different nodes is different.	Operating system on different nodes is different.

Sr. No.	Distributed OS (DOS)	Network OS(NOS)	Middleware based DS
3.	1. As multiple processors present in single machine, number of copies of multiprocessor OS is one. 2. As multiple machines are present multiple (n) copies of multicomputer OS are required.	As multiple machines are present multiple (n) copies of NOS are required.	Middleware is installed on the top of NOS on every machine. Hence multiple machines are preset so multiple copies of OS are required.
4.	1. In multiprocessor OS basis for communication is shared memory. Processors share the memory to communicate with each other. 2. In case of multicomputer OS, Many machines communicate with each other by passing the messages. Message passing is basis for communication	In network OS, basis for communication is files.	In middleware based DS, basis for communication is model specific.
5.	1. In multiprocessor OS resource management is global and central. 2. In multicomputer OS resource management is global and distributed	In network OS resource management is per node. Hence it is easy to scale the system.	Resource management is per node
6.	1. In multiprocessor OS there is no scalability as single machine in which multiple processors are presents 2. Multicomputer OS supports for moderate scalability.	Network operating system offers the scalability.	In middleware based distributed system scalability varies.
7.	Both multiprocessor OS and multicomputer OS are not open. These OS are developed to optimize the performance.	Network operating system is openness. Modern operating systems are built with microkernel design.	It is also open.

1.12 Client Server Model

1.12.1 Clients and Servers

- In client-server model, there are two types of processes. A server process implements the particular type of service. A client process requiring the service implemented by server process sends request to server. Client after sending request waits until reply comes from the server.

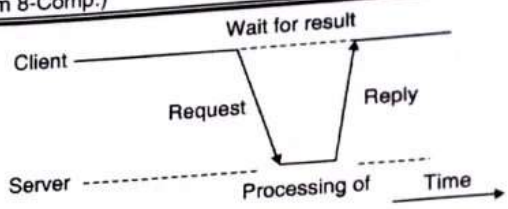


Fig. 1.12.1 : Request-reply based interaction between client and server

- If underlying network is reliable like LAN then connectionless protocol can be used to built client and server. Using connectionless protocol communication is efficient and if there is no loss of packets then request reply interaction work well.0
- Reliable connection oriented protocol cannot be proper solution in LAN but it works well in wide area networks where communication is basically reliable. In connection oriented protocol setting up and terminating the connection is expensive if less number of request and reply messages are exchanged.

1.12.2 Application Layering

Client server application is partitioned between following three layers :

1. User-interface level
2. The processing level
3. The data level

1. User- interface level

- User-interface level is implemented in clients. This level comprises the code that permits the end user to interact with application. The simple user-interface program is character based screen.
- Graphical displays with pop-up and pull down menus can be used at client side. X-Windows-interfaces is one of the example.

2. The processing level

- The core part of the application which operates on database or file system is considered at processing level. In search engines example, user type keyword to be searched is the user interface.
- At back end there is huge database of web pages. The core logic here is to convert the typed keywords by user to database queries. Then ranking the result into list and transforming the list in html pages.

3. Data level

- In client-server model, this level contains programs which maintain the data on which applications perform the operations. This data is persistent.
- Although application is not running, still data remains stored at somewhere.

1.12.3 Client Server Architectures

- The above discussed three levels suggest the different ways to distribute client server application across different machines in the distributed system.
- The simple organization is to keep interface level on client machine and other two levels (processing and data level) on server machine.

Multitiered Architectures

The programs in different levels described above can be distributed among different machines to organizing the client and servers. Following are the different possibilities for two tier architecture :

- Terminal dependent part of the user interface on client machine and user interface application and database on server machine.
- Place entire user interface on client machine and application and database on server machine.
- Place entire user interface and some part of application (front end) on client machine and other part of application and database on server machine. Validation of filled forms is done at client side.
- Place entire user interface and entire application on client machine and database on server machine. In this case client contact only for operations of data to server.
- Place entire user interface and entire application and some part of database (for example cached web pages on local disk) on client machine and database on server machine.

Modern architectures

- One way is vertical distribution where logically different components are placed on different machines. This is just like multitiered architecture. This is related to vertical fragmentation concept in distributed relational databases. In this case, table columns are split and kept on many machines.
- In horizontal distribution, client or server is physically split up in logical parts. Here each part operates on its own share of data set to balance the load. For example, web pages are distributed among many servers and client request is forwarded to these server in round robin fashion.

Review Questions

Q. 1 What is distributed System? What are the main characteristics of distributed system?

Q. 2 Give and explain examples of distributed system.

Q. 3 Write note on "Resource Sharing and Web".

Q. 4 Explain different issues and goals related to design of distributed system.

Q. 5 What is meant by distribution transparency? Explain different types of transparency.

Q. 6 Explain different scaling techniques.

Q. 7 Explain different types of distributed system.

Q. 8 What is open distributed system? Explain in detail.

Q. 9 Explain main types of architectural models of distributed system.

Q. 10 Explain software layers in architectural model of distributed system.

Q.11 What are the different variations in basic client server model of distributed system?

Q. 12 Write note on "interfaces and objects".

Q. 13 What are the design requirements for distributed architectures?

Q. 14 Explain different fundamental models of distributed system.

Q. 15 Explain interaction model of distributed system.

Q. 16 Explain two variants of interaction model.

Q. 17 Write note on "Failure Model".

Q. 18 Explain different types of failures in distributed system.

Q. 19 Write note on "security model" of distributed system.

Q. 20 What is multiprocessor and multicomputer? Explain its basic organization with diagram.

Q. 21 Write note on "Homogeneous Multicomputer System".

Q. 22 Write note on "Heterogeneous Multicomputer System".

Q. 23 What is distributed operating system (DOS)? What are the types of DOS?

Q. 24 Write note on "Distributed Shared Memory Systems (DSMs)"

Q. 25 Explain network operating system. (NOS).

Q. 26 What are the different models of middleware?

Q. 27 Explain different services provided by middleware.

Q. 28 Differentiate between DOS, NOS and middleware based distributed system.

Q. 29 Explain different client server models.

□□□

CHAPTER 2

Module 2

Communication

Syllabus

Layered Protocols, Interprocess communication (IPC): MPI, Remote Procedure Call (RPC), Remote Object Invocation, Remote Method Invocation (RMI), Message Oriented Communication, Stream Oriented Communication, Group Communication.

In distributed system, processes communicate with each other by sending the messages. So it is necessary to study the ways processes exchange the data and information.

2.1 Layered Protocols

Open Systems Interconnection Reference Model (ISO OSI Model) is considered to understand various levels and issues related to communication. Open system communicates with any other open system by using standards rules about format, contents of messages etc. These rules are called as protocols. These protocols can be connection-oriented or connection-less.

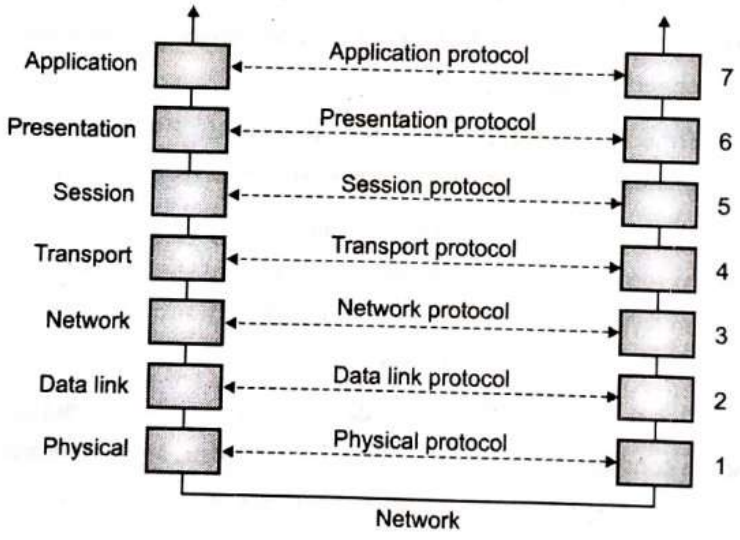


Fig. 2.1.1: Interfaces, Layers and Protocols in OSI model

- In this model total seven layers are present. Each layer provides services to its upper layer. Interface is present between each pair of adjacent layers. Interface defines set of operations through which services is provided to users.
- Application running on sender machine is considered at application layer. It builds the message and handover it to the application layer on its machine. Application layer adds its header at the front of this message and passes to presentation layer which adds its header at the front to the message. Further message is passed down to the session layer which in turn passes the message to transport layer by adding its header and so on.

2.1.1 Lower Layer Protocols

Following is discussion about functions of each layer in OSI model.

- **Physical layer :** This layer deals with issues such as: rate of data transfer per second, in binary form information how much volts to use for 0 and 1, whether transmission between sender and receiver should be simplex, duplex or half duplex. The issues like cables, connectors, number of pins and its meaning are taken care by this layer. The physical layer protocols also deals with standardizing the electrical, mechanical and signaling interface to receive correct information as sent by sender.
- **Data Link Layer (DLL) :** This layer deals with the issues such as:
  - **Framing :** To detect start and end of received of frame.
  - **Flow Control :** This is mechanism controls data transfer rate of sender to avoid load on receiver. DLL implements protocols to get feedback from receiver to sender about slowing down the data transfer rate.
  - **Error Control :** DLL deals with detection and correction of errors in received data by using mechanism such as checksum calculation, CRC etc.
- **Network Layer :** This layer mainly deals with routing. Messages travel through many routers to reach to destination machine. These routers calculate shortest path to forward received message to destination. Internet protocol is most widely used today and it is connectionless. Packets are routed through different paths independent of others. Connection-oriented communication now popular. For example, virtual channel in ATM network.
- **Transport Layer :** This layer is very important to construct network applications. This layer offers all services that are not implemented at interface of network layer. This layer breaks messages coming from upper layer into pieces which are suitable for transmission, assigns them sequence number, congestion control, fragmentation of packets etc. This layer deals with end to end process delivery. Reliability of transport services can be offered or implemented on the top of connection-oriented or connection-less network services.
- In this transport layer, transmission control protocol (TCP) works with IP today in network communication. TCP is connection oriented and user datagram protocol (UDP) is connection-less protocol in this layer.
- **Session and presentation layers :** Session layers deal with dialog control, it keeps track of which parties are currently talking and check pointing to recover from crash. It takes in to account the last checkpoint instead of considering all since beginning. Presentation layer deals with syntax and meaning of transmitted information. It helps the machines with different data representation to communicate with each other.
- **Application Layer :** Initially in OSI layer, electronic mail, file transfer and terminal emulation was considered. Today, all the applications running are considered in application layer. Applications, application specific protocols such as HTTP and general purpose protocol such as FTP is now considered in this layer.

2.2 Interprocess Communication (IPC)

2.2.1 Types of Communication

- **Synchronous Communication :** In this type of communication, client application waits after sending request until reply sent request comes from server application. Example is Remote Procedure Calls (RPC), Remote Method Invocation (RMI).

- **Asynchronous Communication :** In this type of communication, client application continues other work after sending request until reply of sent request comes from server application. RPC can be implemented with this communication type. Other examples can be transferring amount from one account to other, updating database entries and so on.
- **Transient Communication :** In this type of communication, sender application and receiver application both should be running to deliver the messages sent between them. Example is Remote Procedure Calls (RPC), Remote Method Invocation (RMI).
- **Persistent Communication :** In this type of communication, either sender application or receiver application both need not be running to deliver the messages sent between them. Example is email.

2.2.2 Message Passing Interface

- Sockets are considered as insufficient to deal with communication among high-performance multicomputers. It is concluded that, highly efficient applications cannot be implemented with sockets in order to incur minimal cost in communication. Sockets are implemented with general-purpose protocol stack such as TCP/IP which does not suite with high-performance multicomputers. Sockets supports only simple primitives like *send* and *receive*.
- Sockets are not suitable with proprietary protocols developed for high performance interconnection network, for example, used in cluster of workstations (COWs) and massively parallel processors (MPPs).
- Therefore such interconnection network uses proprietary communication libraries which provide high level and efficient communication primitives. Message passing interface (MPI) is specification for users and programmers for message passing libraries. MPI basically is developed for supporting the parallel applications and adapted to transient communication where sender and receiver should be active during communication.
- MPI assumes process crash or network failure as incurable and these failures do not require recovery action. MPI also assumes that known group of processes establishes communication between them. To recognize process group, *groupID* is used and for process, *processID* is used. These are identifiers and the pair (*groupID, processID*) exclusively identifies the sender and receiver of the message. This pair of identifiers excludes using the transport-level of address.

2.3 Remote Procedure Call (RPC)

Program executing on machine A calls procedure or function that is located at machine B is called as Remote Procedure Call (RPC). Although this definition semps to be simple but involves some technical issues. The primitives (*send* and *receive*) used in sending and receiving the messages do not hide communication. In RPC, calling program calls remote procedure by sending parameters in call and callee returns the result of procedure.

2.3.1 RPC Operation

- In Remote Procedure Calls (RPCs) model, a client side process calls a procedure implemented on a remote machine. In this call, parameters are transparently sent to the remote machine (server) where the procedure actually executed. The result of execution then server sent back to the caller. Although called procedure is executed on remotely, it appears as call was executed locally. In this case the communication with remote process remains hidden from calling process.

- While calling a remote procedure, the client program binds with a small library procedure called the client stub. Client stub stands for the server procedure in the client's address space. In the same way, the server program binds with a procedure called the server stub. Fig. 2.3.1 shows the steps in making an RPC.

Step 1 : Client calls the client stub. As client stub is on client machine, this call is a local procedure call (LPC), and the parameters pushed onto the stack in the normal manner.

Step 2 : Client stub packs the parameters into a message and issue a system call to send the message. This process is called marshalling.

Step 3 : Kernel sends the message from the client machine to the server machine.

Step 4 : The kernel on server machine, send the incoming packet to the server stub.

Step 5 : Server stub calls the server procedure.

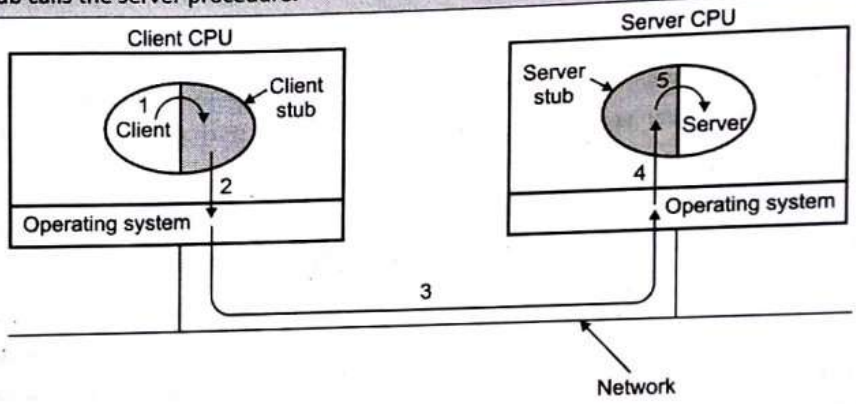


Fig. 2.3.1 : Steps in RPC

- The server then handover the result to server stub which packs it into message. Kernel then send the message to client where kernel at client handover it to client stub. Client stub unpack the message and handover result to client.

2.3.2 Implementation Issues

- Machines in large distributed system are not of same architecture and hence data representation of these machines is different. For example, IBM mainframe uses EBCDIC character code representation, Whereas IBM PC use ASCII code. Hence, passing character code between these pair of machines will not be possible. In the same way, Intel Pentium machines numbers their bytes from MSB to LSB (little endian). SPARK machine stores bytes from LSB to MSB (big endian).
- As client and server applications are in different address space in client and server machine, passing pointer as parameter is not possible. Instead of pointer to data, it is possible to send data as parameter. In C language, array is passed to the function as reference parameter. Hence, it is necessary to send complete array as a parameter to server as array address will be different at server machine.
- Server stub at server machine then creates pointer to this data (passed array from client side) in and passes to server application. Server then sends modified data back to server stub which in turn sends to client stub via kernel. This works for simple data type not for complex data structure like graph. In this example, calling sequence of call by reference is replaced by copy-restore.

- In weakly typed languages like C, it is entirely legal to write a procedure that computes the inner product of two arrays without specifying how large either one is. Each could be terminated by a special value known only to the calling and called procedures. In this case, it is basically impracticable for the client stub to marshal (pack) the parameters as it has no way to know how large they are.
- Sometimes it is difficult to infer the types of the parameters, even from a formal specification or the code itself. Calling *printf* remotely in C language is practically impossible as it takes any number of parameters and of mixed types. If called procedure shifted to remote machine then calling and called procedure cannot share global variables and hence code will fail. By considering all above limitations and taking care of above things, still RPC is used widely

2.3.3 Asynchronous RPC

- Sometimes it is not necessary for client application to block when there is no reply to return from server application. For example, transferring amount from one account to other, updating database entries. In this case, client can continue to carry out its useful work after sending request to server without waiting for reply.
- Asynchronous RPC allows client to continue its work after calling procedure at server. Here, client immediately continue to perform its other useful work after sending RPC request. Server immediately sends acknowledgement to client the moment request is received. Server then immediately calls the requested procedure. After receiving acknowledgement, client continue its work without further blocking. Here client waits till acknowledgement of receipt of request from server arrives.
- In one-way RPC, client immediately continues its work without waiting for acknowledgement from server for receipt of sent request.

2.3.4 DCE RPC

- Distributed Computing Environment RPC is one of the examples of RPC system developed by Open Software Foundation (OSF). Its specifications have been adopted in Microsoft base system for distributed computing. DCE is true middleware system initially designed for UNIX. Now, it has been ported to all major operating systems.
- DCE uses client server model as programming model where user processes are clients to access services remotely from server processes at server machine. Some services are in built in DCE and other are implemented by application programmers. Client and server communication takes place through RPC.

  Services provided by DCE are :

  - **Distributed File Service :** It offers transparent way to access any file in the system in same manner.
  - **Directory service :** It keeps track on all resources distributed worldwide including machines, printers, data, files, servers and many more.
  - **Security Service :** It offers access to the resources to only authorised persons.
  - **Distributed Time Service :** To synchronize the clocks of different machines.

Goals and Working of DCE RPC

- RPC system allows client to call local procedure in order to access remote service from server. RPC system can automatically locate the server and establishes connection between client and server application (binding).

- It can fragment and reassemble the messages, handle transport in both directions and converts data types between client and servers if machine on which they running have different architecture and hence, different data representation.
- Clients and servers can be independent of one another. Implementation of client and servers in different languages is supported. OS and hardware platform can be different for client and server applications. RPC system hides all these dissimilarities.
- DCE RPC system contains several components such as libraries, languages, daemons, utility programs etc. Interface definitions are specified in interface definition language (IDL). IDL files contains type definitions, constant declarations, function prototypes, and information to pack parameter and unpack the result. Interface definitions contains syntax of call not its semantics.
- Each IDL file contains globally unique identifier for specified interface. This identifier client sends to server in first RPC message. Server then checks its correctness for binding purpose otherwise it detects error.

Steps in Writing a Client Server Application in DCE RPC

- First, call *uuidgen* program to generate prototype IDL file which contains unique interface identifier not generated anywhere by this program. This unique interface identifier is 128 bit binary number represented in ASCII string in hexadecimal. It encodes location and time of creation to guarantee the uniqueness.
- Next step is to edit this IDL file. Write the remote procedure names and parameters in file. This IDL file then is compiled by using IDL compiler. After compilation, three files are generated: header file, Client stub and server stub.
- Header file contains type definitions, constant declarations, function prototypes and unique identifier. This file is included *(#include)* in client and server code. The client stub holds procedures which client will call from server.
- These procedure collects and marshals parameters and convert it outgoing message. It then calls runtime system to send this message. Client stub also is responsible for unmarshaling the reply from server and delivering it to client application. The server stub at server machines contains procedures which are called by runtime system there when message from client side arrives. This again calls actual server procedure.
- Write server and client code. Client code and client stub both are compiled to convert it in object files which are linked with runtime library to produce executable binary for client. Similarly at server machine, server code and server stub both are compiled to convert it in object files which are linked with runtime library to produce executable binary for server.

Binding Client to Server in DCE RPC

- Client should be able to call to server and server should accept the clients call. For this purpose, registration of server is necessary. Client locates first server machine and then server process on that machine.
- Port numbers are used by OS on server machine to differentiate incoming messages for different processes. DCE daemon maintains table of *server-port number* pairs. Server first asks the OS about port number and registers this end point with DCE daemon. Server also registers with directory service and provide it network address of server machine and name of server. Binding a client to server is carried out as shown in Fig. 2.3.2.

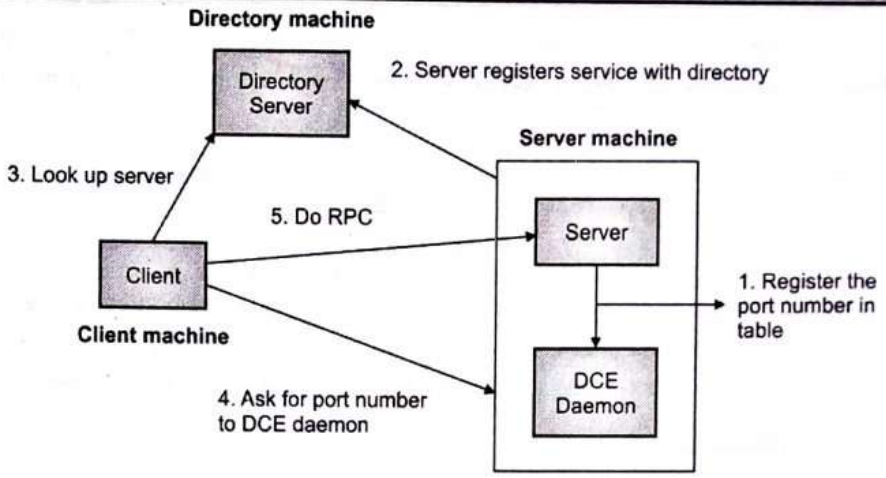


Fig. 2.3.2

- Client passes name of server to directory server which returns network address of server. Client then contact DCE daemon to get port number of server running on server machine. Once client knows both network address and port number, RPC takes place.

2.4 Remote Object Invocation

Objects hides its internal details from external world by means of well-defined interface. CORBA and DCOM are the examples of object based distributed systems.

2.4.1 Distributed Objects (RMI : Remote Method Invocation)

- Objects encapsulates data, called the state and operations on those data, called as methods. Methods are invoked through interface. Process can access or manipulate state only through object's interface. Single object may implements several interfaces. Similarly, for given interface definition, several objects may provide implementation for it.

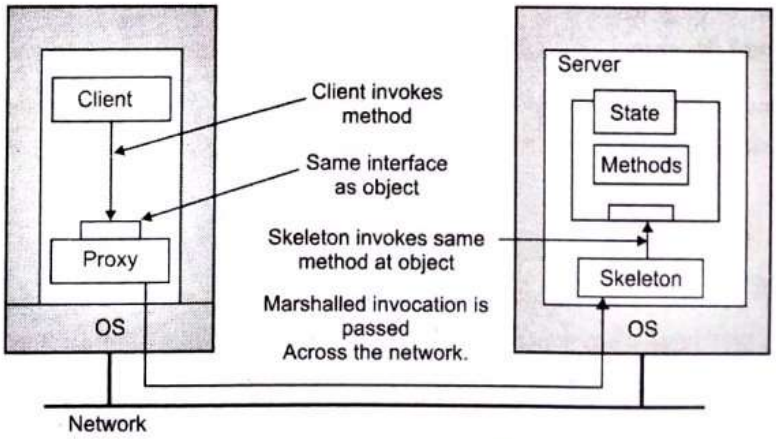


Fig. 2.4.1: Organization of Remote object with proxy at client side

- We can keep object's interface on one machine and object itself on other machine. This organization is referred as distributed object. Implementation of object's interface is called as proxy. It is similar to client stub in RPC and remains in client address space.
- Proxy marshals clients method invocation in message and unmarshal reply messages which contains result of method invocation by client. This result, proxy then returns to client. Similar to server stub in RPC, skeleton is present on server machine
- Skeleton unmarshals incoming method invocation requests to proper method invocations at the object's interface at the server application. It also marshals reply messages and forward them to client side proxy. Objects remains on single machine and their interfaces only made available on different machines. This is also called as remote object.

Compile-time Versus Runtime objects

- Language-level objects called as compile-time objects. These objects are supported by object oriented languages such as Java, C++ and other object oriented languages. Object is an instance of class. Compile time objects makes it easy to build the distributed applications in distributed system.
- In Java, objects are defined by means of class and interfaces that class implements. After compiling interfaces, we get server stub and client stub. The generated code after compiling the class definition permits to instantiate the java objects. Hence, Java objects can be invoked from remote machine. The disadvantage of the compile-time objects is its dependency on particular programming language.
- Runtime objects are independent of programming language and explicitly constructed during run time. Many distributed object-based system uses this approach and allows construction of applications from objects written in multiple languages. Implementation of runtime objects basically is left open.
- Object adapter is used as wrapper around implementation so that it appears to be object and its methods can be invoked from the remote machine. Here, objects are defined in terms of interfaces they implements. This implementation of interface then registered with object adapter which makes available the interface for remote invocation.

Persistent and Transient Objects

- Persistent object continue to exist although server exits. Server at present managing persistent object stores object state on secondary storage before it exit. If newly started address needs this object then it reads object state from secondary storage.
- In contrast to persistent object, transient object exists till server manages that object exists. Once server exit, the transient object also exit.

Object References

- Client invokes the method of remote object. The client binds with an object using object reference which must contain sufficient information to permit this binding. Object reference includes network address of machine on which object is placed, plus port number (endpoint) of server assigned by its local operating system.
- The limitation of above approach is that, when server crashes, it will be assigned different port number by OS after recovery. In this case, all the object references become invalid.

- The solution to this problem to have local daemon process on machine which listens to a well-known port number and records *server-to-port number* assignment in endpoint table. In this case, server ID is encoded in object reference is used as index into endpoint table. While binding client to object, daemon process is asked for server's current port number. Server should always register with local daemon.
- The problem with encoding network address in object reference is that, if server moves to other machine then again all the object references becomes invalid. This problem can be solved by keeping location server which will keep track on machine where server is currently running. In this case, object reference contains network address of location server and systemwide identifier for server.
- Object reference may also include more information such as identification of protocol that is used to bind with object and the protocol supported by object server, for example TCP or UDP. Object reference may also contain implementation handle which refers to complete implementation of proxy which client can dynamically loads for binding with object.

Parameter Passing

- Objects can be accessed from remote machines. Object references can be used as parameter to method invocation and these references are passed by value. As a result object references can be copied from one machine to the other. If process has object reference then it can bind to the object whenever required.
- It is not efficient to use only distributed or remote objects. Some objects such as Integers and Booleans are small. Therefore it is important to consider references to local objects and remote objects. For remote method invocation, object reference is passed as parameter if object is remote object. This reference is copied and passed as value parameter. Whereas, if object is in the same address space of client then entire object is passed along with the method invocation. That means, object is passed by value.
- As a example, suppose client code is running on Machine 1 having reference to local object O1. Server code is running on Machine 2. Let the remote object O2 resides at machine 3. Suppose client on Machine 1 calls server program with object O1 as a parameter to call. Client on machine 1 also holds reference to remote object O2 at machine 3. Client also uses O2 as a parameter to call. In this invocation, copy of object O1 and copy of the reference to O2 is passed as parameter while calling server on Machine 2. Hence, local objects are passed by value and remote objects are passed by reference.

Example : Java RMI

In Java, distributed objects have been integrated in language itself and goal was to keep semantics as much of the nondistributed objects. The main focus is given on high degree of distribution transparency.

The java Distributed Object Model

- Java supports distributed object as a remote object which resides on remote machines. Interfaces of the remote objects is implemented by means of proxy which is local object in client interface. Proxy offers precisely same interface as remote object.
- Cloning of object creates exactly same copy of object and its state. Cloning the remote object. Cloning the remote object also requires cloning of proxies which currently are bound with object. Therefore cloning is carried out only by server which creates exact copy of object in server address space. In this case, no need to clone proxies of actual object, only client has to bind to cloned object in order to access it.

- In Java, method can be declared as *synchronized*. If two processes try to call *synchronized* method at the same time then only one process is permitted to proceed and other is blocked. In this manner, access to object's internal data is entirely serialized.
- This blocking of process on synchronized method can be at client side or server side. At client side, client is blocked in client-side stub which implements object interface. As every client on other machines are blocked at client side, synchronization among them is required which is complex to implement.
- Server side blocking of client process is possible. But, if client crashes while server is executing its invocation then protocol is needed to handle this situation. In Java RMI blocking on remote object is restricted to proxies.

Java Remote Object Invocation

- In Java, any primitive or object type can be passed as parameter to RMI if it can be marshaled. Then it is said that the objects are serializable. Most of the object types are serializable except platform dependent objects. In RMI local objects are passed by value and remote objects are passed by reference.
- In Java RMI, object reference includes network address, port number of server and object identifier. Server class and client class are two classes used to build remote object. Server class contains implementation of remote object that runs on server. It contains description object state, implementation of methods that works on this state. Skeleton is generated from interface specification of the object.
- Client class contains implementation of client-side code. This class contains implementation of proxy. This class is generated from interface specification of the object. Proxy builds the method invocation in message which then it sends to server. The reply from server is converted in result of method invocation by proxy. Proxy stores network address of server machine, port number and object identifier.
- In Java, proxies are serializable. Proxies can be passed to remote process in terms of message. This remote process then can use this proxy for method invocation of remote object. Proxy can be used as reference to remote object. Proxy is treated as local object and hence can be passed as parameter in RMI.
- As size of proxy is large, implementation handle is generated and used to download the classes to construct proxy. Hence, implementation handle replaces marshaled code as part of remote object reference. Java virtual machine is on every machine. Hence, marshaled proxy can be executed after unmarshaling it on remote or other machine.

2.5 Message-Oriented Communication

In Remote Procedure Call (RPC) and Remote method invocation (RMI), communication is inherently synchronous where client blocks till reply from server arrives. For communication between client and server, both sending and receiving sides should be executing. Messaging system assumes both side applications are running while communication is going on. Message queuing-system permits processes to communicate although receiving side is not running at the time communication is initiated.

2.5.1 Persistence Synchronicity in Communication

- In message-oriented communication, we assume that applications executes on hosts. These hosts offer interface to the communication system through which messages are sent for transmission.

- All the hosts are connected through network of communication servers. These communication servers are responsible for routing the messages between the hosts connected in network.
- Buffer is available at each host and communication server. Consider Email system with above configuration of hosts and communication servers. Here, user agent program runs on host using which users can read, compose, sends or receive the messages. User agent sends message permits to send message at destination.
- User agent program at host submits message to host for transmission. Host then forward this message to its local mail server. This mail server stores the received message in output buffer and look up transport level address of destination mail server. It then establishes the connection with destination mail server and forward message by removing it from output buffer.
- At destination mail server, message is put in input buffer in order to deliver to designated receiver. User agent at receiving host can check incoming messages on regular basis by using services of interface available there. In this case, messages are buffered at communication servers (mail servers).
- Message sent by source host is buffered by communication server as long as it is successfully delivered to next communication server. Hence, after submitting message for transmission, sending application need not continue execution as message is stored by communication server. Also, it is not necessary for receiving application to be executing when message was submitted. This form of communication is called as **persistent communication**. Email is example of **persistent communication.**
- In **transient communication**, communication system (communication server in above discussion) stores the message if both sender and receiver applications are executing. Otherwise message is discarded. In the example discussed above, if communication server is not able to deliver message to next communication server or receiver then message is discarded.
- All the transport-level communication services offer transient communication and in this communication router plays the role of communication server. Router simply drops message if it is not able to forward it to next router or destination host.
- **In asynchronous communication**, sender continues execution after submitting message for transmission. This sent message either remains in buffer of sending host or at first communication server. In **synchronous communication**, sender is blocked until message is stored in local buffer of destination host or delivered to receiving application.

2.5.2 Combination of Communication Types

- Following are the combinations of different types of communication
- **Persistent Asynchronous Communication:** In this type of communication, message is persistently stored in buffer of local host or in buffer of first communication server. Email is the example of persistent asynchronous communication. Fig. 2.5.1 shows this type of communication in which process A continues its execution after sending the message. After this, B receives the message when it will start running.

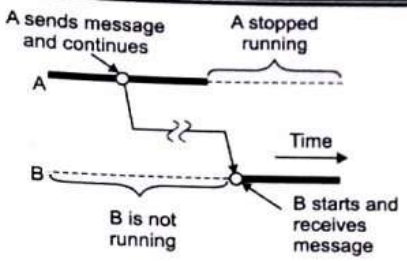


Fig. 2.5.1

- **Persistent Synchronous Communication :** In this communication sender is blocked until its message is stored at receiver's buffer. In this case, receiver side application may not be running at the time message is stored. In Fig. 2.5.2 Process A sends message and is blocked till message is stored in buffer of receiver. Note that, process B starts running after some time and receives the message.

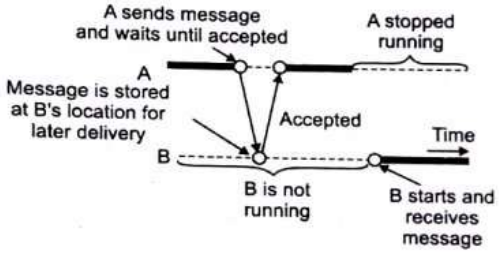


Fig. 2.5.2

- **Transient Asynchronous Communication :** In this type of communication, sender immediately continues its execution after sending the message. Message may remain in local buffer of sender's host. Communication system the routes the message to destination host. If receiver process is not running at the time message is received then transmission fails. Example is transport-level datagram services such as UDP. In Fig. 2.5.3, Process A sends message and continues its execution. While at receiver host, if process B is executing at the time message is arrived then B receives the message. Example is one way RPC.

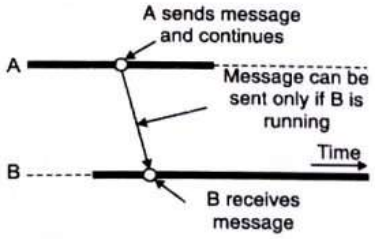


Fig. 2.5.3

- **Receipt-Based Transient Synchronous Communication:** In this type of communication, sender is blocked until message is stored at local buffer of receiver host. Sender continues its execution after it receives acknowledgement of receipt of message. It is shown in Fig. 2.5.4.

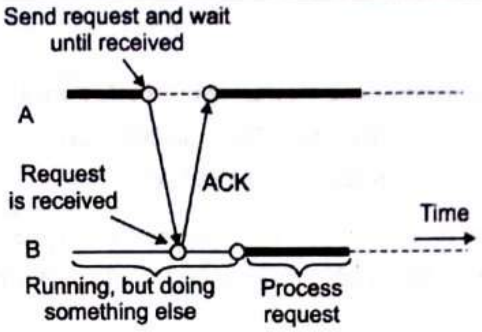


Fig. 2.5.4

- **Delivery-Based Transient Synchronous Communication :** In this type of communication, sender is blocked until message is delivered to the target process. It is shown in Fig. 2.5.5. Asynchronous RPC is example of this type of communication.

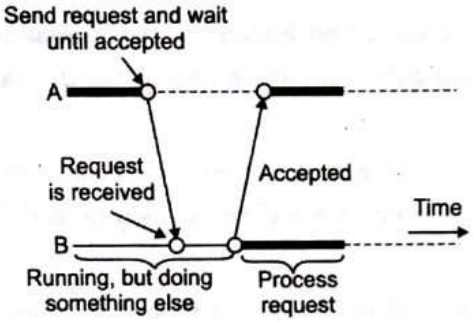


Fig. 2.5.5

- **Response-Based Transient Synchronous Communication:** In this type of communication, sender is blocked until response is received from receiver process. It is shown in Fig. 2.5.6. RPC and RMI sticks to this type of communication.

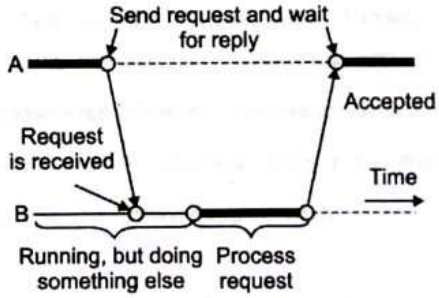


Fig. 2.5.6

- In distributed system, there is need of all communication types as per requirements of applications to be developed.

2.5.3 Message-Oriented Transient Communication

Transport layer offers message-oriented model using which several applications and distributed systems are built as per needs. Berkley sockets are transport-level sockets which is the example of messaging.

Berkley Sockets

- Transport layer offer programmers to use all the messaging protocols through simple set of primitives. This is possible due to focus given on standardizing the interfaces. The standard interfaces also allow to port the application on different computers. Socket interface is introduced in Berkley UNIX.
- Socket is communication endpoint to which application writes data to be sent to applications running on other machines in the network. Application also reads the incoming data from this communication endpoint. Following are socket primitives for TCP.
- **Socket :** This primitive is executed by **server** to create new communication endpoint for particular transport protocol. The local operating system internally reserve resources for incoming and outgoing messages for this specified protocol. This primitive is also executed by **client** to create socket at client side. But, here binding of local address to this socket is not required as operating system dynamically allocate port when connection is set up.
- **Bind :** This primitive is executed by **server** to bind IP address of the machine and port number to created socket. Port number may be possibly well-known port. This binding informs the OS that server will receive message only on specified IP address and port number.
- **Listen : Server** calls this primitive only in connection-oriented communication. This call permits operating system to reserve sufficient required buffers for maximum number of connections that caller wanted to accept. It is nonblocking call.
- **Accept :** This is executed by server and call to these primitive blocks the caller until connection request arrives. Local operating system then creates new socket with same properties like original one. This new socket is then returned to the caller which permits the server to fork off the process to handle actual communication through this new connection. After this, server goes back and waits for new connection request on original socket.
- **Connect :** This primitive executed by **client** to get transport-level address in order to send connection request. The client is blocked till the connection is set up successfully.
- **Send :** This primitive executed by both **client** and **server** to send the data over the established connection.
- **Receive :** This primitive executed by both **client** and **server** to receive the data over the established connection.
- Both client and server exchange the information through **write** and read primitive which establish sending and receiving of the data.
- **Close: Client** and **server both calls** close primitive to close the connection.

Message Passing Interface

- The message passing interface (MPI) is discussed in section 2.2.2. It uses messaging primitive to support transient communication. Following are some of the messaging primitives used.
- **MPI_bsend :** This primitive is executed by sender of the message to put outgoing message to local send buffer of MPI runtime system. Here, sender continues after copying message in buffer. Transient Asynchronous Communication in Fig. 2.5.3 is supported through this primitive.

- **MPI_send :** This primitive is executed by sender to send the message and then waits either until message is copied in local buffer of MPI runtime or until receiver has initiated receive operation. First case is given by Fig. 2.5.4 and second case is given in Fig. 2.5.5.
- **MPI_ssend :** This primitive is executed by sender to send the message and then waits until receipt of the message starts by receiver. This case is given in Fig. 2.5.5.
- **MPI_sendrecv :** This primitive supports synchronous communication given in Fig. 2.5.6 Sender is blocked until reply from receiver.
- **MPI_isend :** This primitive avoids the copying of message to MPI runtime buffer from user's buffer. Sender passes pointer to message and after this MPI runtime system handles the communication.
- **MPI_issend :** Sender passes pointer to MPI runtime system. Once message processing is done by MPI runtime system , sender guaranteed that receiver accepted the message for processing.
- **MPI_recv :** It is called to receive a message. It blocks caller until message arrives.
- **MPI_irecv :** Same as above but here receiver indicates that it is ready to accept message. Receiver checks whether message is arrived or not. It supports asynchronous communication.

2.5.4 Message-Oriented Persistent Communication

Message-queuing systems or Message-oriented Middleware (MOM) supports persistent asynchronous communication. In this type of communication, sender or receiver of the message need not be active during message transmission. Message is stored in intermediate storage.

Message-Queuing Model

- In message-queuing system, messages are forwarded through many communication servers. Each application has its own private queue to which other application sends the message. In this system, guarantee is given to sender that its sent message will be delivered to recipient queue. Message can be delivered at any time.
- In this system, receiver need not be executing when message arrives in its queue from sender. Also, sender need not be executing after its message is delivered to the receiver. In exchange of messages between sender and receiver, message is delivered to receiver with following execution modes.
  - Sender and receiver both are running.
  - Sender running but receiver passive.
  - Sender passive but receiver running.
  - Both sender and receiver are passive.
- System wide unique name is used as address for destination queue. If message size is large then underlying system fragments and assembles the message in the manner transparent to communicating applications. This leads to having simple interface to offer to applications. Following are the primitives offered.
  - **Put :** Sender calls this primitive to pass the message to system in order to put in designated queue. This call is non-blocking.
  - **Get :** It is blocking call by which process having rights removes message from queue. If queue is empty then process blocks.

- **Poll :** It is nonblocking call and process executing it polls for expected message. If queue is empty or expected message not found then process continues.
- **Notify :** With this primitive, installed handler is called by receiver to check queue for message.

General Architecture of a Message-Queuing System

- In the message-queuing system, source queue is present either on the local machine of the sender or on the same LAN. Messages can be read from local queue. The message put in queue for transmission contains specification of destination queue. Message-queuing system provides queues to senders and receivers of the messages.
- Message-queuing system keeps mapping of queue names to network locations. A queue manager manages queues and interacts with applications which sends and receives the messages. Special queue managers work as routers or relays which forward the messages to other queue managers. Relays are more suitable as in several message-queuing systems, dynamically mapping of queue-to-location not available.
- Hence, each queue manager should have copy of queue-to-location mapping. For larger size message-queuing system, this approach leads to network management problem. To solve these problems, only routers need to be updated after adding or removal of the queues.
- Relays thus provide help in construction of scalable message-queuing systems. Fig. 2.5.7 shows relationship between queue-level-addressing and network-level-addressing.

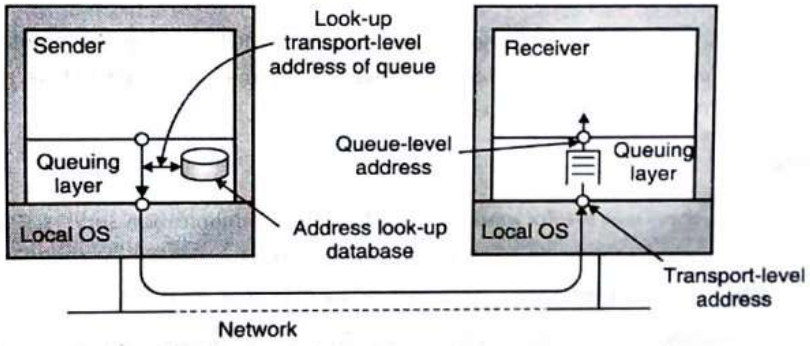


Fig. 2.5.7

Message Brokers

- It is required for each application that is added in message-queuing system to understand the format of messages received from other applications. This will leads to fulfil the goal of integration of new and old applications into single coherent distributed system.
- For diverse applications, consideration of sequence of bytes is more appropriate to achieve above goal. In message-queuing systems, message brokers nodes which are application level gateway, is dedicated for conversion of messages. It converts incoming messages in the formats understandable by destination applications. Message brokers are external to the message-queuing systems and not integral part of it.
- In more advance settings of conversion, some information loss is expected. For example, conversion between X.400 and internet email messages. Message broker maintains database of rules to convert message of one format into other format. These rules are manually inserted in this database.

Example : IBM's WebSphere Message-Queuing System

- MQSeries is IBM's WebSphere product which is now known as WebSphere MQ. Fig. 2.5.8 shows General organization of IBM's message-queuing system. Queue managers manage all queues and extract messages from its send queue. It also forward the messages to other queue managers. It pick up the incoming messages over network and put them in appropriate incoming queues.

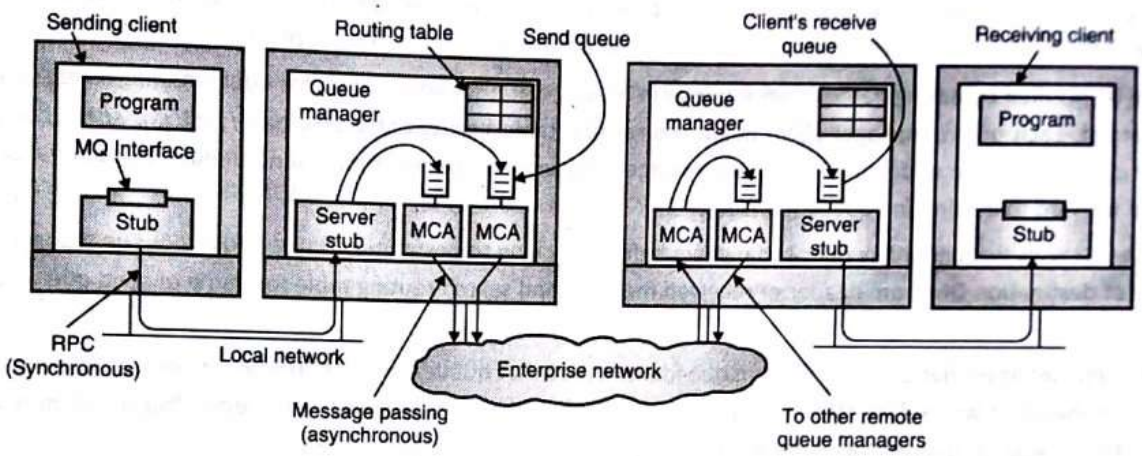


Fig. 2.5.8 : General organization of IBM's message-queuing system

- Maximum default size of message is 4 MB but can be increased up to 100 MB. Normal queue size is 2 GB but it can be of more size depending on underlying operating system. Connection between two queue managers is through message channels, which is an idea of transport-level connections. A message channel is a unidirectional, reliable connection between queue managers which acts as sender and receiver of messages. Through this connection, queued messages are transported.
- Message channel agent (MCA) manages each of the two ends of message channel. MCA at sender side checks send queue for message. If found, it wraps message in transport level packet and send it to receiving MCA along the connection. On the other hand, receiving MCA waits for incoming packet and unwraps it and store it in appropriate queue.
- Queue manager and application can be linked into the same process. Application communicates with queue manager through standard interface. In other organization, queue manager and application can be kept on separate machines.
- Each message channel has exactly one send queue. It fetches the message from this queue to send it at receiver end. If sender and receiver MCA are up and running then transfer along the channel is possible. Both MCAs can be started manually. Application can also activate sender or receiver MCA in order to start its end of channel.
- In other case, sending MCA is started by setting off the trigger when message is put in send queue. This trigger is associated with handler to start the sending MCA. A control message can be send to start the other end MCA when one end is already started. Following are some attributes associated with message channel agents. These attribute value should be compatible between sending and receiving MCA and its negotiation takes place prior to setting up the channel.
  - **Transport types :** Determines transport protocol to be used.
  - **FIFO delivery :** Delivery of messages will be in FIFO manner.

- **Message length :** Maximum length of message.
- **Setups retry count :** Set up maximum number of retries to start up the receiving MCA.
- **Delivery retries :** Maximum times MCA try to put received message into queue.

Message Transfer

- A message should carry a destination address to send message by sending queue manager to receiving queue manager. The first part of the address consists of the name of the receiving queue manager. The second part is the name of the destination queue of queue manager to which the message is to be appended.
- Route is specified by having name of send queue in message. It also means to which queue manager message is to be forwarded. Each queue manager (QM) maintains routing table having entry as a pair *(destQM, sendQ)* in it. In the entry *destQM* is name of destination queue manager and *sendQ* is name of local send queue in which message to be put to forward to destination queue manager.
- Message travels through many queue managers before reaching to destination. Intermediate queue manager extracts name of destination QM from header of received message and search routing table for name of send queue to append message to it.
- Each queue manager has a systemwide unique identifier for that queue manager. This identifier is unique name to the queue manager. If we replace QM then applications sending messages may face problems. This problem is solved by using a local alias for queue manager names.

2.6 Stream-Oriented Communication

In much communication, applications expect incoming information to be received in precisely defined time limits. For example: audio and video streams in multimedia communication. Many protocols are designed to take care of this issue. Distributed system offers exchange of such time dependent information between senders and receivers.

2.6.1 Continuous Media Support

Representation of information in storage or presentation media (monitor) is different. For example, text information is stored in ASCCI or in Unicode format and images can be in GIF or JPEG format. Same is true for audio information. The media is categorized as continuous and discrete representation media. In continuous media, relationship about time between different data items matters to infer actual meaning of it. In discrete media, timing relationship doesn't matter between different data items.

Data Stream

- Sequence of data units called as data streams. It is applied to both continuous and discrete media. Transmission modes differ in terms of timing aspects for continuous and discrete media.
- In **asynchronous transmission mode,** timing constraints on received data items doesn't matter. For example, in file transfer data items received as data streams is irrelevant to its transmission time by sender.
- In **synchronous transmission mode,** maximum end to end delay is defined for every unit in a data stream. Data units in a data stream can also be received at any time within defined maximum end to end delay. It will cause no harm at receiver side. It is not important here that, data units in stream can be sent faster than maximum tolerated delay.
- In **isochronous transmission mode,** maximum and minimum end to end delay (bounded jitter) is defined for every unit in a data stream. This mode is important in distributed multimedia system to represent audio and video.

- **Simple stream** contains only single sequence of data. **Complex stream** consist of many such simple streams (substreams). These substreams need synchronization between them. Time dependent relationship is present between substreams in complex stream. Example of complex stream is transferring movie. This stream comprises one video stream, two streams to transfer movie sound in stereo. The other stream may be subtitles to display for deaf.
- Stream is virtual connection between source (sender) and sink (receiver). These source and sink could be process or device. For example, process in one machine reading data byte by byte from hard disk and sending it across network to another process on other machine. In turn, this receiver process may deliver received bytes to local device.
- In multiparty communication, data stream is multicast to many sinks. To adjust with requirements of quality of service for receivers, a stream is configured with filters.

2.6.2 Streams and Quality of Service (QoS)

- It is necessary to preserve temporal relationship in a stream. For continuous data stream, timeliness, volume and reliability decides quality of service. There are many ways to state QoS requirements.
- A flow specification contains precise requirements for QoS such as bandwidth, transmission rate, delay etc. Token bucket algorithm generates tokens at constant rate. A bucket (buffer) holds these tokens which represent number of bytes application can send across the network. If bucket is full then tokens are dropped. Application has to remove tokens from buffer to pass the data units to network. Application itself may remain unknown about its requirements.

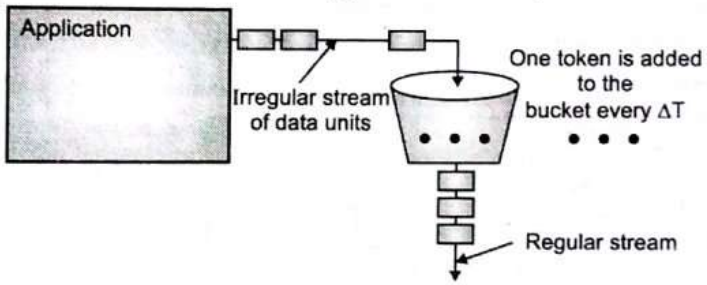


Fig. 2.6.1: Token Bucket Algorithm

Following are the service requirement and characteristics of input specified in flow specification.

- **Characteristics of Input**
  - **Maximum data unit size in bytes:** It defines maximum size of data units in bytes.
  - **Token bucket rate in bytes/sec:** To permit the burstiness, as application is permitted to pass complete bucket to network in single operation.
  - **Token bucket size in bytes**
  - **Maximum transmission rate in byte/sec:** This is to limit the rate of transmission to specified maximum in order to avoid extreme burst.
- **Service Requirements**
  - **Loss sensitivity (bytes) and Loss interval (micro second) :** Both collectively defines maximum acceptable loss rate.
  - **Burst loss sensitivity (data units) :** Number of data units my lost.
  - **Minimum delay noticed (micro second) :** Defines delay parameter that network can delay to deliver data before receiver notice it.
  - **Maximum delay variation (micro second) :** Maximum tolerated jitter.

- **Quality of guarantee :** If number is low then no problem. But for high number, if network cannot give guarantee then system should not establish the stream.

Setting up a Stream

- There is no single best model to specify QoS parameters and to describe resources in network. These resources are bandwidth, processing capacity, buffers. Also, there is no single best model to translate these QoS parameters to resource usage. QoS requirements are dependent on services that network offers.
- **Resource reSerVation protocol (RSVP)** is transport-level protocol which reserves resources at router for continuous streams. The sender handover the data stream requirements in terms of bandwidth, delay, jitter etc to RSVP process on the same machine which then stores it locally.
- RSVP is receiver initiated protocol for QoS requirements. Here, receiver sends reservation request along the path to the sender. Sender in RSVP set up path with receiver and gives flow specification that contains QoS requirements such as bandwidth, delay, jitter etc to each intermediate node.
- Receiver when is ready to accept incoming data, it handover flow specification (reservation request) along the upstream path to the receiver: may be reflecting lower QoS parameter requirement.
- At sender side, this reservation request is given by RSVP process to admission control module to check sufficient resources available or not. Same request is the passed to policy control module to check whether receiver has permission to make reservation. Resources are then reserved if these to tests are passed. Fig. 2.6.1 shows organization of RSVP for resource reservation in distributed system.

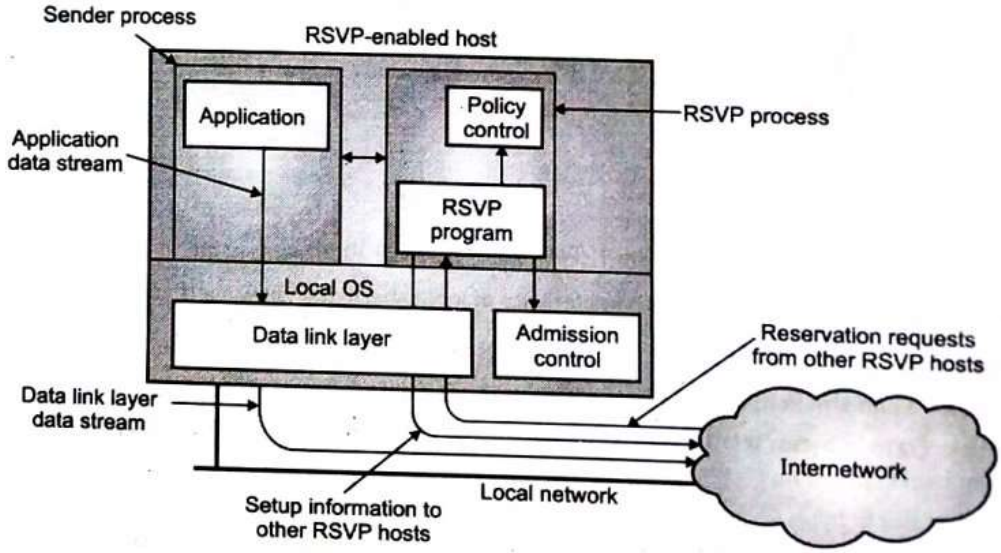


Fig. 2.6.2 : RSVP Protocol

- Besides above network protocol, internet classifies the data in terms of differential services which decides priority of packets to be forwarded by current router. Forwarding class is also provided in which traffic is divided in four classes and three ways to drop packets if network is congested. In this way applications may distinguish time-sensitive packets from non significant ones.

- Distributed system also offers buffers to reduce jitter. The delayed packets are stored in buffer at receiver side for maximum time which will be delivered to application at regular rate. Error detection and correction techniques also used for which allows retransmission of erroneous packets. The lost packets can be distributed over time while delivering to applications.

2.6.3 Stream Synchronization

- In case of complex stream, it is necessary to maintain temporal relations between different data stream. Consider example of synchronization between discrete data stream and a continuous data stream. Web server stores slide show presentation which contains slides and audio stream. Slides come from server to client in terms of discrete data stream. The client should play the same audio with respect to current slide. Audio stream here gets synchronized with slide presentation.
- In playing a movie, video stream need to be synchronized with audio stream. The data units in different streams actually are synchronized. The meaning of data unit depends on level of abstraction at which data stream is considered. It is necessary to display video frames at a rate of 25 Hz or more. If we consider broadly used NTSC standard of 29.97 Hz, we could group audio samples into logical units that last as long as a video frame is displayed (33 msec).

Synchronization Mechanisms

- For actual synchronization we have to consider following two issues.
  - Mechanisms to synchronize two streams.
  - Distribution of these mechanisms in network environment.
- At lowest level, synchronization is carried out on data items of simple streams. In this approach, application should implement synchronization which is not possible as only it has low-level facilities available. Other better alternative could be to offer an application an interface which permits it to control stream and devices in simple way.
- At receiver side of complex stream, different substreams needs to be synchronized. To carry out synchronization, a synchronization specification should be available locally. Common approach is to offer this information implicitly by multiplexing the different streams into a single stream having all data units, including those for synchronization. MPEG streams are synchronized in this manner.
- Another important issue is whether synchronization should handle by sender or receiver. In case of sender side synchronization, it may be possible to merge streams into a single stream with a different type of data unit.

2.7 Group Communication

- In distributed system, it is necessary to have support for sending data to multiple receivers. This type of communication is called multicast communication. To support multicast communication, many network-level and transport-level solutions have been implemented. The issues in these solutions were to setup the path to disseminate the information.
- Alternative to these solutions is application level multicasting techniques as peer to peer solutions are usually deployed at application layer. It now became easier to set up communication paths.

- Multicast messages offer good solution to build distributed system with following characteristics.
  - **Fault tolerance based on replicated services :** Replicated service is available on group of servers. Client requests are multicast to all these servers in group. Each server performs the same operation on this request. If some member of group fails, still client will be served by other active members (servers).
  - **Finding discovery servers in spontaneous networking :** Servers and clients can use multicast messages to find discovery services to register their interfaces or to search the interfaces of other services in the distributed system.
  - **Better performance through replicated data :** Replication improves performance. If in some cases, replica of data is placed in client machine then changes in data item is multicast to all the processes managing the replicas.
  - **Event Notification :** Multicast to group to notify if some event occurs.

2.7.1 Application-Level Multicasting

- In application-level multicasting, nodes are organized into an overlay network. It is then used to disseminate information to its members. Group membership is not assigned to network routers. Hence, network level routing is the better solution with compare to routing messages within overlay.
- In overlay network nodes are organized into tree, hence there exists a unique path between every pair of nodes. Nodes also can be organized in mesh network and hence, there are multiple paths between every pair of nodes which offers robustness in case of failure of any node.

Scribe : an Application Level Multicasting Scheme

- It is built on top of Pastry which is also a DHT (distributed hash table) based peer-to-peer system. In order to start a multicast session, node generates multicast identifier *(mid)*. It is randomly chosen 160 bit key. It then finds *SUCC(mid)* which is node accountable for that key and promotes it to be the root of the multicast tree that will be used to send data to interested nodes.
- If node X wants to join tree, it executes operation *LOOKUP(mid)*. This lookup message now with request to join MID (multicast group) gets routed to *SUCC(mid)* node. While traveling towards the root, this join message passes many nodes. Suppose this join message reaches to node Y. If Y had seen this join request for *mid* first time, it will become a forwarder for that group. Now node X will become child of Y whereas the latter will carry on to forward the join request to the root.
- If the next node on the root, say *Z* is also not yet a forwarder, it will become one and record Y as its child and persist to send the join request. Alternatively, if Y (or *Z)* is already a forwarder for *mid,* it will also note the earlier sender as its child (i.e., *X* or Y, respectively), but it will not be requirement to send the join request to the root anymore, as Y(or *Z*) will already be a member of the multicast tree.
- In this way, multicast tree across the overlay network with two types of nodes gets created. These nodes are : pure forwarder that works as helper and nodes that are forwarders as well, but have clearly requested to join the tree.

Overlay Construction

- It is not easier to build efficient tree of nodes in overlay. Actual performance is merely based on the routing of messages through the overlay. There are three metric to measure quality of an application-level multicast tree. These are link stress, stretch, and tree cost. **Link stress** is per link and counts how frequently a packet crosses the same link.

- If at logical level although packet is forwarded along two different connections, the part of those connections may in fact correspond to the same physical link. In this case, link stress is greater than 1. The **stretch or Relative Delay Penalty** (RDP) measures the ratio in the delay between two nodes in the overlay and the delay that those two nodes would experience in the underlying network. **Tree cost** is a global metric, generally related to minimizing the aggregated link costs.

2.7.2 Gossip-Based Data Dissemination

- As there are large number of nodes in large distributed system, epidemic protocols can be used effectively to disseminate the information among many nodes. There is no centralized component to coordinate the information dissemination. Only local information can be used for the same.
- In order to understand the principle of these algorithms, consider all updates for a specific data item are initiated at a single node. Because of the same, write-write can be avoided.

Information Dissemination Models

- Theory of epidemics studies the spreading of infectious diseases. This basic principle is used by epidemic algorithms to disseminate the information among nodes in network. The node having data to spread to other node is called as infected node. If up till now any node has not seen this data then it is considered as susceptible node. If the node is already updated and it is not willing to spread its data to other nodes is called as removed.
- The famous propagation model is that of anti-entropy. In this mode, node X randomly selects node Y to exchange the updates with it. These updates exchanges takes place with one of the following three different approaches.
  - Node X only pushes its own updates to node Y.
  - Node X only pulls in new updates from node Y.
  - Push-pull approach in which both nodes X and Y sends updates to each other.
- If quick spreading of updates is required then using push-based approach of spreading updates by infected node will be bad option. If infected nodes are many in numbers then probability of selecting the susceptible node by each infected node will be relatively less. As a result, a particular node may remains susceptible for a longer period as it is not selected by an infected node.
- In case of many infected nodes, the pull-based approach is better option. In this case, susceptible nodes contact to infected node in order to pull the updates. The susceptible nodes will also become infected one.
- If only a single node is infected, spreading of updates will spread rapidly across all nodes using either push or pull approach. In this case push-pull can also be the best approach. Consider round as time span involved in which every node will at least once have taken the initiative to exchange updates with a randomly chosen other node. Then, to propagate single update to all nodes O(logN) rounds are required. In this case, N is number of nodes in the system.
- The variation of above approach is **gossiping** or **rumor spreading**. In this approach, suppose node X has just updated data item x then it contact any node Y randomly and try to push the update to it. In the meantime, if node Y is already updated by any other node then it becomes removed node. Gossiping approach rapidly spreads news. In this approach still there will be some nodes which will remain ignorant.

- In large number of nodes which takes part in epidemic, suppose fraction S of nodes that remain ignorant of update (susceptible) satisfies following equation.

$$s = e^{-(k+1)(1-s)}$$

- For k = 3, s is less than 0.02. Scalability is main advantage of epidemic algorithms.

Removing Data

- It is hard for epidemic algorithms to spread the deletion of data item. It is because the deletion destroys information related to deleted data item. In this case if data item from node is suppose removed, then node will receive old copies of data item and it will be considered as new one.
- The solution to above problem is to consider deletion of data item as another update and keep record of it. In this way, old copies will be treated as versions that have been updated by delete operation and not something new. The recording of deletion is executed by spreading the death certificates.
- Each node may slowly build huge database of death certificates of deleted data items. The solution to this problem is to create the death certificate with timestamp. The death certificates then can be removed after updates propagate to all the nodes within known finite time which is maximum elapsed propagation time.
- A few numbers of nodes (For example, say node X) still maintains this death certificate. This is for those nodes to which previous death certificate was not reached. If node X has death certificate for data item x and update come to node X for the same data item x. In this case, node X will again spread the death certificate of data item x.

Applications of Epidemic Protocols

- Providing positioning information about nodes can help in constructing specific topologies.
- Gossiping can be used to find out nodes that have a small number of outgoing wide-area links.
- Collecting or actually aggregating information.

Review Questions

Q. 1 Explain ISO OSI model of communication?

Q. 2 Write short note on "Message Passing Interface".

Q. 3 What is remote procedure call (RPC)? Write steps in RPC operation.

Q. 4 What are the issues in implementation of remote procedure call (RPC)? Explain.

Q. 5 Write short note on "Asynchronous RPC".

Q. 6 Write short note on :DCE RPC".

Q. 7 What are the goals of DCE RPC?

Q. 8 Explain steps in writing a client server application in DCE RPC.

Q. 9 Explain binding of client to server in DCE RPC.

Q. 10 What is remote method invocation (RMI)? Explain distributed object model.

Q. 11 What is the role of proxy and skeleton in RMI?

Q. 12 What is the role of client stub and server stub in RPC?

Q. 13 Write note on "Compile-time Versus Runtime objects".

Q. 14 Write note on "Compile-time Versus Runtime objects".

Q. 15 Write note on "Persistent and Transient Objects".

Q. 16 What is object reference? Explain.

Q. 17 Write short note on "Java RMI"

Q. 18 Explain parameter passing in RMI.

Q. 19 Explain JAVA distributed object model.

Q. 20 Define synchronous, asynchronous, transient and persistent communication.

Q. 21 Explain with neat diagram different types of communication.

Q. 22 Explain Berkley Sockets.

Q. 23 Explain MPI primitives.

Q. 24 Explain Message-Queuing Model.

Q. 25 Explain general architecture of a message-queuing system.

Q. 26 What is role of message broker in message-queuing system? Explain.

Q. 27 Explain in detail IBM's WebSphere Message-Queuing System.

Q. 28 What is data stream? What are the different transmission modes?

Q. 29 What is discrete and continuous media.

Q. 30 Write note on "Streams and Quality of Service (QoS)"

Q. 31 Explain working of RSVP protocol to set up the stream

Q. 32 Explain stream synchronization in detail.

Q. 33 Explain Application-Level Multicasting.

Q. 34 Write short note on "group communication".

Q. 35 Explain Gossip-Based Data Dissemination in multicast communication.

❑❑❑

Synchronization

Syllabus

Clock Synchronization, Logical Clocks, Election Algorithms, Mutual Exclusion, and Distributed Mutual Exclusion-Classification of mutual Exclusion Algorithm, Requirements of Mutual Exclusion Algorithms, and Performance measure. Non Token based Algorithms : Lamport Algorithm, Ricart–Agrawala's Algorithm, Maekawa's Algorithm, Token Based Algorithms : Suzuki-Kasami's Broardcast Algorithms, Singhal's Heurastic Algorithm, Raymond's Tree based Algorithm, Comparative Performance Analysis.

Introduction

- Apart from the communication between processes in distributed system, cooperation and synchronization between processes is also important. Cooperation is supported through naming which permits the processes to share resources. As an example of synchronization, multiple processes in distributed system should agree on some ordering of events occurred.
- Processes should also cooperate with each other to access the shared resources so that simultaneous access of these resources will be avoided. As processes run on different machines in distributed system, synchronization is no easy to implement with compare to uniprocessor or multiprocessor system.

3.1 Clock Synchronization

- In centralized system, if process P asks for the time to kernel, it will get it. After sometime if process Q asks for time, naturally it will get time having value greater than or equal to the value of time of process P received. In distributed system, agreement on time is important and necessary to achieve it.
- Practically, different machines clock in distributed system differs in their time value. Therefore, clock synchronization is required in distributed system. Consider example of UNIX make program. Editor is running on machine A and Compiler is running on machine B. Large UNIX program is collection of many source files. If few files suppose modified then no need to recompile all the files in program. Only modified files require recompilation.
- After changing source files, programmer starts make program which checks timestamp of creation of source and object file. For example xyz.c has timestamp 3126 and xyz.o has timestamp 3125. In this case, since object file xyz.o has greater timestamp, make program confirms that xyz.c has been changed and recompilation is required. If abc.c has timestamp 3128 and abc.o has 3129 then no compilation is required.
- Suppose there is no global agreement on time. In this case suppose abc.o has timestamp 3129 and abc.c is modified and assigned time 3128.

- This is due to slightly lagging of machine clock on which abc.c is modified. Here make program will not call compiler and object file abc.o will be old version although its source file abc.c is changed. The resulting executable binary of large size UNIX program will contain mix of object files from old and new sources.

3.1.1 Physical Clocks

- Every machine has timer which is machined quartz crystal. This quartz crystal oscillates when kept under tension. The frequency of oscillation depends on type of crystal, how it is cut and how much tension is applied. Two registers, **counter** and **holding register** is associated with crystal. Counter value gets decremented by one after completion of one oscillation of the crystal. When counter value gets to 0 then interrupt is generated. Again counter gets reloaded from holding register.
- Each interrupt is called **clock tick** and it is possible to program a timer to generate 60 clock ticks per second. Battery backed-up CMOS RAM keeps time and date. After booting, with each clock tick interrupt service procedure adds 1 to current time. Practically it is not true that crystal in all the machines runs with same frequency. The difference between two clocks time value is called **clock skew**.
- In real time system, actual clock time is important. So consideration of multiple physical clocks is necessary. How to synchronize these clocks with real world clocks and how to synchronize clocks with each other are two important issues that need to be considered.
- Mechanical clock was invented in 17^{th} century and since then time has been measured astronomically. When sun reaches at its highest apparent point in sky is called as **transit of sun.** Solar day is interval between two consecutive transits of sun. Each day have 24 hour. 1 hour contain 60 minutes and in each minute contain 60 seconds. Hence, each day have (24x60x60) total 84600 seconds. **Solar second** is $1/84600^{th}$ solar day.
- In 1940 it was invented that earth is slowing down and hence period of earth's rotation is also varying. As a results, astronomers considered large number of days and taken the average of it before dividing by 84600. This is (1/84600) is called **mean solar second.**
- In 1948, physicist started measuring time with cesium 133 clock. One mean solar second is equal to time that is needed to cesium 133 atom to make 9,192631,770 transitions. Now days, 50 laboratories worldwide kept cesium clock and reports periodically time to BIH in Paris which takes average of all the time called as **International atomic time (TAI).** Now 84600 TAI seconds is about 3 msec less than mean solar day. BIH solved this problem by using leap second when difference between TAI and solar time grows 800 msec. After correction of time, it is called **universally coordinated time (UTC).**
- National institute of standard time (NIST) run shortwave broadcast radio station that broadcast short pulse at start of each UTC second with accuracy of ± 10 msec. In practice, this accuracy is no better than ±10msec. Several earth satellites also provide UTC service.

3.1.2 Global Positioning System (GPS)

- GPS was launched in 1978. It is a satellite-based distributed system which has been used mainly for military applications. Now days it is used in many civilian applications such as traffic navigation and in GPS phones.

- GPS makes use of 29 satellites which travels in an orbit at a height of about 20,000 km. Every satellite has 4 atomic clocks which are calibrated regularly from special stations on earth. Each satellite constantly broadcast its current position. This message is with timestamp as per its local time.
- The receiver of this message on earth can accurately calculate its actual position by using three satellites. Three satellites are required to resolve the longitude, latitude, and altitude of a receiver on Earth. Practically it is not true that all clocks are absolutely synchronized. Satellite message also takes some time to reach to receiver.
- It may also happen that, receiver clock is lagging or leading with respect to clock time of satellite. GPS does not consider the leap second. Hence, measurement is not perfectly accurate.

3.1.3 Clock Synchronization Algorithms

- If one machine is with UTC time then all other machines clocks time should be synchronized with respect to this UTC time. Otherwise, all the machines clocks together should be with same time. Consider timer of each machine interrupts X times a second. Assume that value of this clock is Y. When UTC time is t then value of clock at machine p is $Y_p(t)$. If all the clocks value is UTC time t then $Y_p(t) = t$ for all p, which is required. It means, dY/dt should be 1.
- In real world, timer do not interrupt exactly X times a second. If X=0 then timer should generate 216,000 ticks per hour. Practically, machine gets this value in the range 215,998 to 216,002 ticks per hour. For any constant α, if $1-\alpha \leq dY/dt \leq 1+\alpha$ then timer is working within its specification. The constant α is specified by manufacturer of timer and called as **maximum drift rate.**
- If dY/dt is greater than 1 then clock is faster. If dY/dt is equal to 1 then clock is perfect clock. If dY/dt is less than 1 the clock is slower with respect to UTC time.

Cristian's Algorithm

- In this algorithm, it is assumed that one machine (time server) has WWV receiver. Hence, this machine clock value is UTC time. All other machines (client) synchronize their clock with this UTC time. Let C_{UTC} is the time replied by time server.

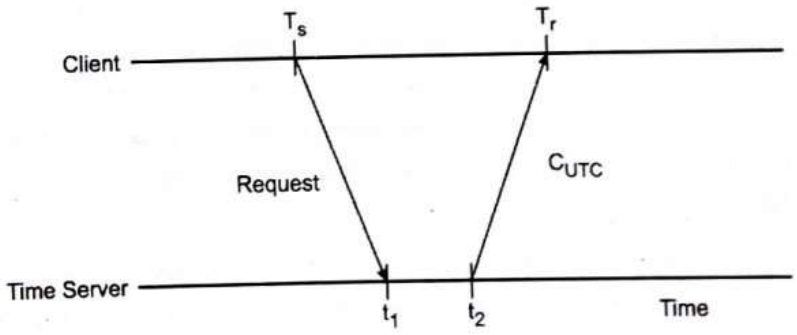


Fig. 3.1.1

- When client receives response from time server at time Tr, it sets its clock at time C_{UTC}. This algorithm has two problems, one is major and other is minor problem. Major problem occur when clients clock is fast. The received C_{UTC} will be less than client's current value C.

- In UNIX program example explained previously, if on client machine object file is compiled just after clock change will have timestamp less than the source which was modified just before clock change.
- The solution to this problem is to introduce the change slowly. If each interrupt of timer adds 5 msec to the time then add 4 msec to time until time gets corrected. Time can be advanced by adding 6 msec for each interrupt instead of forwarding all at once.
- As a minor problem, it takes some time to reach the reply from time server to client. This propagation time is (Tr – Ts)/2. Add this time to the C_{UTC} time which is replied by server to set the client's clock time. Let time elapsed between t_1 and t_2 is interrupt processing time (I) at server. Then message propagation time is (Tr – Ts – I). One way propagation time will be the half of this time. In this algorithm, time server is passive as it replies as per query of client.

The Berkeley Algorithm

- In Berkeley UNIX, time daemon (server) is active as it polls every machine asking current time there. It then calculates average time of received answers from all machines. It then tells other machines either advance their clocks to new time or slow their clocks until specified time is set.
- This algorithm is used in system where WWV receiver is not there for UTC time. The daemon time is periodically set manually by operator.

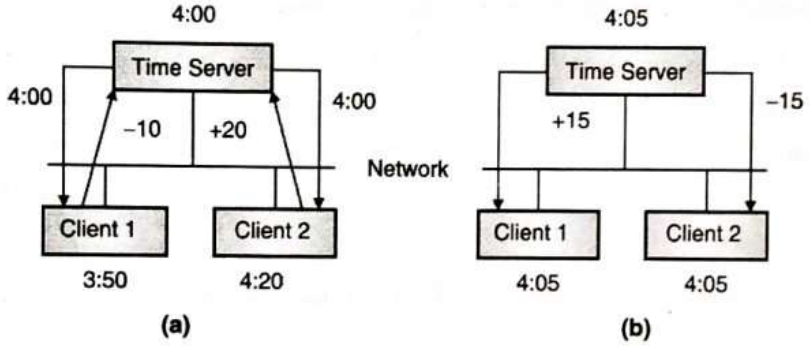


Fig. 3.1.2

- As shown in Fig. 3.1.2(a), initially time daemon has time value of 4:00. Client 1 has clock time value 3:50 and client 2 has clock time value of 4:20. Time server asks to other machines about their current time including it. Client 1 replies that, it is lagging by 10 with respect to server time which is 4:00. Similarly, client 2 replies that, it is leading by 20 with respect to server time which is 4:00. Server will reply as 0 to itself.
- Meanwhile server time increases to 4:05. Therefore server sends client 1 and 2 messages about how to adjust their clocks. It tells client 1 to advance clock time by 15. Similarly, it tells to client 2 to decrement the time value by 15. Server itself tells to add 5 time its own clock time. It is shown in Fig. 3.1.2(b).

Averaging Algorithms

- Cristian algorithm and Berkeley algorithm both are centralized algorithms. Centralized algorithms have certain disadvantages. Averaging algorithm is decentralized algorithm. One of these algorithms works by dividing time in fixed length intervals. Let T0 is any agreed upon moment in past. The k^{th} interval start at time T0 +kS and runs until T0 + (k+1)S. Here S is a system parameter.

- At the start of each interval, every machine broadcast its clock time. This broadcast will happen at different time at different clocks as machines clock speed can be different. After machine finish the broadcast, it starts local timer in order to collect broadcast from other machines in some time interval I. When all the broadcasts are collected at every machine, following algorithms are used.
  - Average the time values collected from all machines.
  - Other version is to discard n highest and n lowest time values and take the average of rest.
  - Another variation is to add propagation time of message from source machine in received time. Propagation time can be calculated using known topology of the network.

Clock Synchronization in Wireless Networks

- In wireless network, algorithm needs to be optimized considering energy consumption of nodes. It is not easy to deploy time server just like traditional distributed system. So, design of clock synchronization algorithms requires different insights.
- Reference broadcast synchronization **(RBS)** is a clock synchronization protocol. It focuses on internal synchronization of clocks just like the Berkeley algorithm. It does not consider synchronization with respect to UTC time. In this protocol, only receiver synchronizes while keeping sender out of loop.
- In sensor network, when message leaves the network interface of sender, it takes constant time to reach destination. Here multi-hop routing is not assumed. Propagation time of the message is measured from the point message leaves the network interface of the sender. RBS considers only delivery time at receivers. Sender broadcasts reference message say m. When receiver x receives this message, it records time of receiving of m which is say $T_x m$. This time is recorded with its local clock.
- Two nodes x and y exchanges their receiving time of m to calculate the offset. M is total number of sent reference messages.

$$\text{Offset } [x, y] = \frac{\sum_{k=1}^{M} (Tx, k - Ty, k)}{M}$$

- As clocks drift apart from each other, calculating average only will not work. Standard linear regression is used to calculate offset function.

Network Time Protocol (NTP)

- In cristian algorithm when time server send reply message with UTC time, the problem is to find actual propagation time. The propagation time of reply will definitely affect the reporting of actual UTC time.
- When reply will reach at client, message delay will have outdated reported time. The good estimation for this delay can be calculated as shown in following Fig. 3.1.3.

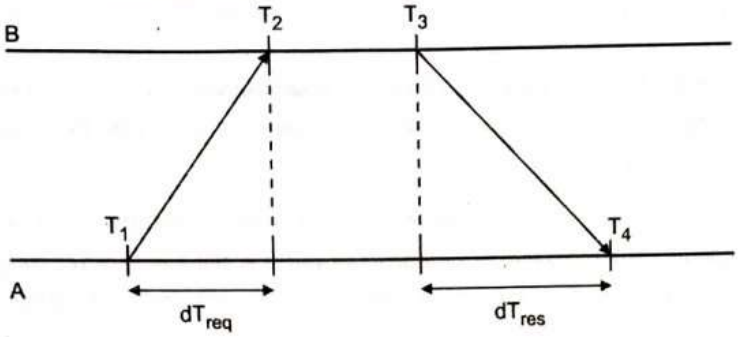


Fig. 3.1.3

- A sends message to B containing timestamp T_1. B also records its receiving time T_2 by its local clock. B then send reply containing timestamp T_3 along with its recorded timestamp T_2. Finally, A records time T_4 at which response arrives. Let $T_2 - T_1 = T_4 - T_3$. Estimation of its offset by A is given as below.

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

- The value of the θ will be less than zero if A's clock is fast. In this case A has to set its clock to backward time. Practically it will produce problem like object file is compiled just after clock change will have timestamp less than the source which was modified just before clock change. The solution to this problem is to introduce the change slowly. If each interrupt of timer adds 5 msec to the time then add 4 msec to time until time gets corrected. Time can be advanced by adding 6 msec for each interrupt instead of forwarding all at once.
- In case of NTP, this protocol is set up pair-wise between servers. In other word, B will also query to A for its current time. Along with calculation of offset as above, estimation of μ for delay is carried out as

$$\mu = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

- Eight pairs of (θ, μ) values are buffered. Smallest value for μ is the best estimation for the delay between the two servers. Subsequently the associated value of θ is the most reliable estimation of offset. Some clocks are more accurate, say B's clock. Servers are divided in strata.
- If A contacts B, it will only correct its time provided its own stratum level is higher than that of B. After synchronization, A's stratum level will become one higher than that of B. Because of the symmetry of NTP, if A's stratum level was lower than that of B, B will adjust itself to A. Stratum-I server is with WWV receiver or with cesium clock. Many features of NTP are related to identification and masking of errors and security attacks.

3.2 Logical Clocks

- In many applications, all machines should agree on same time although this time does not agree with UTC time or real time. In this case, internal consistency of clocks is essential. Many algorithm works based on internal consistency of clocks and not with real time. For these algorithms, clocks are said to be logical clocks.
- Lamport showed that, if two processes are not interacting then lack of their clocks synchronization will not cause any problem. He also pointed out that, processes should agree on the order in which events occur and not on exactly what time it is.

3.2.1 Lamport's Logical Clocks

- For synchronization of logical clocks, Lamport defined **happen-before** relation. The expression x → y is read as "x happens before y". This means all processes agree that first event x occur and after x, event y occur. Following two situations indicates the happen before relation.
  - If x and y are the events in same process and event x occur before y then x → y is true.
  - If x is event of sending the message by one process and y is the event of receiving same message by other process then x → y is true. Practically, message takes nonzero time to deliver to other process.
- If x → y and y → z then x → z. It means happen before relation is transitive. If any two processes does not exchange messages directly or indirectly, then x → y and y → x where x and y are the events that occur in these processes. In this case, events are said to be **concurrent**.
- Let T(x) be the time value of event x. If x → y then T(x) < T(y). If x and y are the events in same process and event x occur before y then x → y then T(x) < T(y). If x is event of sending the message by one process and y is the event of receiving same message by other process then x → y then all the processes should agree on the values T(x) and T(y) with T(x) < T(y). Clock time T assumed to go forward and should not be decreasing. Time value can be corrected by adding positive value.
- In Fig. 3.2.1 (a), three processes A, B and C are shown. These processes are running on different machines. Each clock runs with its own speed. Clock has ticked 5 times in process A, 7 times in process B and 9 times in process C. This rate is different due to different crystal in timer.

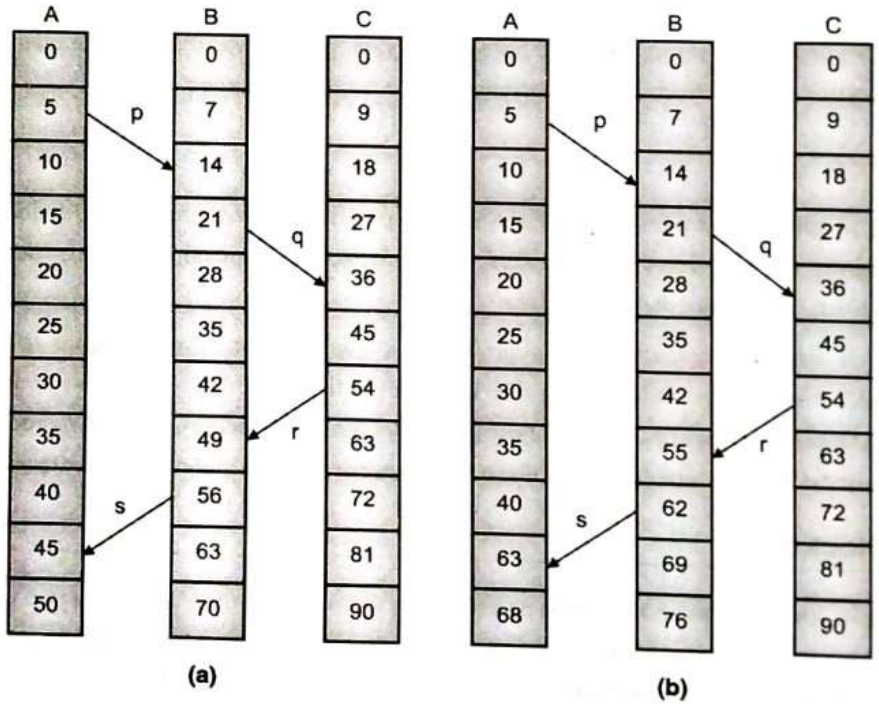


Fig. 3.2.1

- At time 5, process A sends message p to process B and it is received at time 14. Process B will conclude that, message has taken 9 ticks to travel from process A to process B, if message contains sending time. Same is true about message q from process B to process C.
- Process B sends message s at time 56 and it is received by process A at time 45. Similarly, Process C sends message r at time 54 and it is received by process A at time 49. This should not happen and it should be prevented. It shows sending time is larger than receiving time.
- Message r from process C leaves at time 54. As per happen before relation, it should reach at process B at time 55 or later. Same is true for message s which should reach to process A at time 63 or later. Receiver fast forwards the clock by one more than the sending time. That is why the sender always sends sending time in message. This is shown in Fig. 3.2.1 (b). If two events occur in sequence, then clock must tick at least once. If process sends and receive message in quick succession, then clock should be advanced at least by 1 between these two send and receive events.
- If two events occur at the same time in different processes, then it should be separated by decimal point. If such events occur at time 20 then former should be considered at 20.1 and later at 20.2.
  - If $x \rightarrow y$ in the same process then $T(x) < T(y)$.
  - If x is event of sending the message y is the event of receiving the message then $T(x) < T(y)$.
  - For all distinguishing events x and y, $T(x) \neq T(y)$.
- Total ordering of all the events can be carried out with above algorithm.

3.2.2 Application of Lamport Timestamp : Total Order Multicasting

- Ordering of events between communicating processes is very important to achieve consistency at different sites. For example, two servers in different cities A and B maintains bank database. For example, customer accounts balance initially is Rs 1000. If customer deposited Rs 100 in city A and at the same time branch manager gives 1% interest on account balance in city B.
- These are two events related to same copy of database that is present on two different servers. Lets us say these events as event A (updating account balance by adding Rs 100) and event B (updating account balance by adding 1% interest). If these events executes in different order at these databases then inconsistency in account balances would occur.
- In city A, if event A is executed first and then suppose event B is executed. In this case account balance in city A will be updated as : 1000+100 = 1100 and then adding Rs 11 interest = **Rs 1111**. In city B, if event B is executed first and then suppose event A is executed. In this case, account balance in city B will be updated as: 1000+10 = 1010 and then adding Rs 100 = Rs **1110.**
- These Event A and B messages arrive at both sites A and B in order explained above due to communication delay in network. If clocks of all machines in network show same time then ordering of messages can be done as per timestamp when message was sent. Practically, it is not possible.
- Lamport proposed notion of logical time to arrange the messages at different sites in same order for execution. **Totally order multicast** is multicasting operation in which all messages are delivered to receiver in same order.

- Suppose group of processes multicast messages to each other. Sender of the message also receives its own sent message. Each message carries its sending time. Assume all the processes are receiving messages in the order they were sent and three is no loss of messages.
- Each process sends acknowledgement (ACK) for the received message. Hence as per Lamport algorithm, timestamp of the ACK will be greater than the timestamp of received message. Here Lamport's clock ensures that, no any two messages will have same timestamp. Timestamps also gives global ordering of events.

3.2.3 Vector Clocks

- In Lamport's clock, if $x \rightarrow y$ then $T(x) < T(y)$. But, it does not tell about relationship between event x and y. Lamport clocks do not capture causality.
- Suppose user and hence processes join discussion group where processes post articles and post reaction to these posted articles. Posting of articles and reactions are multicast to all the members of the group.
- Consider the total order multicasting scheme and reactions should be delivered after their corresponding postings. The receipt of article should be causally preceded by posting of reaction. Within group, receipt of reaction to an article must always follow the receipt of that article. For independent articles or reactions, order of delivery does not matter.
- The causal relationship between messages is captured through vector clocks. Event x is said to be causally precede event y, if $VT(x) < VT(y)$. Here, VT(x) is a vector clock assigned to event x and VT(y) is vector clock assigned to event y.
- It is assumed that, each process P_i maintains a vector V_i having following properties.
  - $VT_i[i]$ maintains number of events that have taken place at process P_i.
  - If $VT_i[j] = k$ then P_i is aware about k number of events occurred at process P_j.
- Each time event occurs at process P_i, $VT_i[i]$ gets incremented by 1 and it ensures to satisfy first property. Whenever process P_i sends the message m to other process, it sends its current vector with timestamp v_t along with it (piggybacking), which ensures to satisfy second property said above.
- Because of this, receiver knows the number of events occurred at other processes before receipt of message m from process P_i and on which message m may causally depends.
- When process P_j receives message m, it modifies each entry in its table to maximum of the $VT_j[k]$ and vt[k]. The vector at P_j now shows the number of messages it should receive to have at least seen the same messages that preceded the sending of message m. After this every VTj[j] is incremented by 1.

3.3 Election Algorithms

- To fulfill the need of processing in distributed system, many distributed algorithms are designed in which one process acts as coordinator to play some special role. This can be any process. If this process fails, then some approach is required to assign this role to other process. In this case, election is required to elect the coordinator.
- In group of processes, if all processes are same then the only way to assign this responsibility is on the basis of some criterion. This criterion can be the identifier which is some number assigned to process. For example, it could be network address of machine on which process is running. This assumption considers that only one process is running on the machine.

- Election algorithm locates process with highest number in order to elect it as a coordinator. Every process is aware about the process number of other processes. But, processes do not know about which processes are currently up or which ones are currently crashed. Election algorithm ensures that, all processes will agree on newly elected coordinator.

3.3.1 Bully Algorithm

- In this algorithm, if any process finds that the coordinator is not responding to its request, then it initiates the election. Suppose, process P initiates the election. Election is carried out as follows :
  - Process P sends **Election** message to all the process having higher number than it.
  - If no one replies to **Election** message of P then it wins the election.
  - If anyone higher number process responds then it takes over. Now P's work is done.
- Any process may receive **Election** message from its lower-numbered colleague. Receiver replies with **OK** message to sender of **Election** message. This reply conveys the message that, receiver is alive and will take over the further job. If receiver is currently not holding an election, it starts election process.
- Eventually all processes will quit except the highest number process. This process wins the election and send **Coordinator** message to all processes to convey the message that he is now new coordinator.
- In Fig. 3.3.1, initially process 17 (higher-numbered) is coordinator. Process 14 notices that, coordinator 17 has just crashed as it has not given response to request of process 14. Process 14 now initiates election by sending **Election** message to process 15, 16 and 17 which are higher-numbered processes.

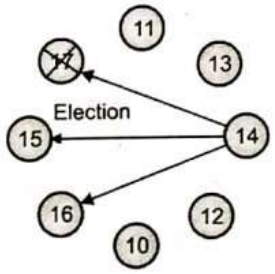


Fig. 3.3.1

- As shown in Fig. 3.3.2, process 15 and 16 responds with **OK** message. Process 17 is already crashed and hence does not send reply with **OK** message. Now, job of process 14 is over.

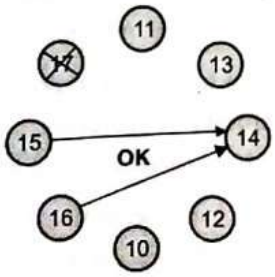


Fig. 3.3.2

- Now process 15 and 16 each hold an election by sending **Election** message to its higher-numbered processes. It is shown in following Fig. 3.3.3 (a). As process 17 is crashed, only process 16 replies to 15 with **OK** message. This is shown in Fig. 3.3.3 (b).

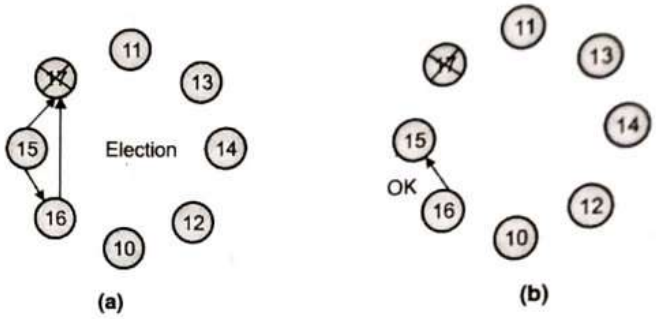


Fig. 3.3.3

- Finally, process 16 wins the election and send **coordinator** message to all the processes to inform that, it is now new coordinator as shown in Fig. 3.3.4. In this algorithm, if two processes detect simultaneously that the coordinator is crashed then both will initiate the election. Every higher-number process will receive two **Election** messages. Process will ignore the second **Election** message and election will carry on as usual.

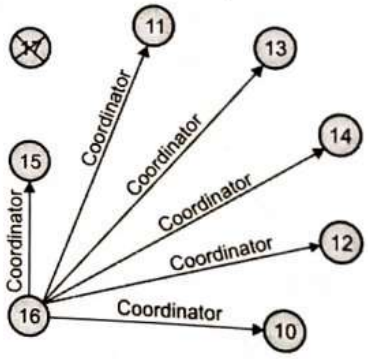


Fig. 3.3.4

3.3.2 Ring Algorithm

- This ring algorithm does not use token and Processes are physically or logically ordered in ring. Each process has its successor. Every process knows about who is its successor. When any process notices that coordinator is crashed, it builds the **Election** message and sends to its successor. This message contains the process number of sending process.
- If successor is down then process sends message to the next process along the ring. Process locates next running process along the ring if it finds many crashed processes in sequence. In this manner, the receiver of **Election** message also forwards the same message to its successor by appending its own number in message.
- In this way, eventually message returns back to the process that had started the election initially. This incoming message contains its own process number along with process numbers of all the processes that had received this message.

- At this point, this message gets converted to **coordinator** message. Once again this **coordinator** message is circulated along the ring to inform the processes about new coordinator and members of the ring. Off course, the highest process number is chosen from the list for new coordinator by process which has started the election.

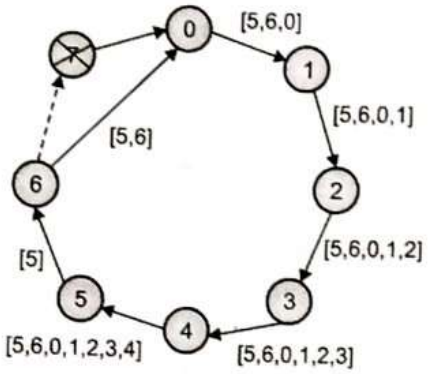


Fig. 3.3.5

- In Fig. 3.3.5, process 5 notices crash of coordinator which was initially process 7. It then sends **Election** message to process 6. This message contains number of the process 5. As process 7 is crashed, process 6 forward this message to its new successor process 0 and append its number in the same list. In this way message is received by all the processes in ring.
- Eventually, message arrives at process 5 which had initiated the election. Highest number in this list is 6. So message is again circulated in previous manner to inform all the processes that, process 6 is now new coordinator.

3.3.3 Elections in Wireless Networks

- Distributed system may also have nodes connected through wireless networks. In this system, election of coordinator is carried out with first step as build network phase followed by election. Let the wireless ad hoc network includes nine nodes P, Q, R, S, T, U, V, W, X and Y each having capacity in the range 0 to 8.
- It is assumed that node cannot move. Let p is the source node which initiates the election. There are minimum of two nodes connected to each node as shown in Fig. 3.3.6. Values within the circle show capacity of respective node. Total numbers of interconnections are kept less to have minimum network traffic. Let P be the source node which initiates election.

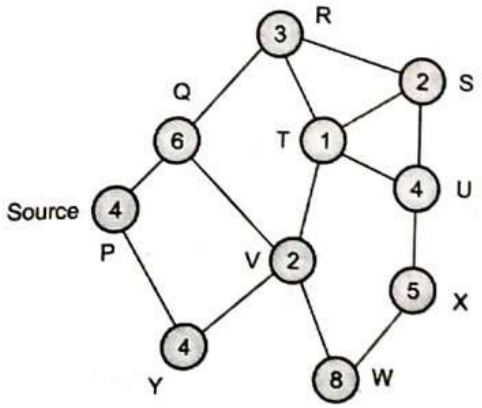


Fig. 3.3.6

- Let any node (consider, source as node A) initiates election and send **Election** message to its immediate neighbors which are in its range. When any node receives this **Election** message first time, it chooses sender of message as its parent. The receivers then send out this received **Election** message to its immediate neighbors which are in its range except to its chosen parent. When node receives **Election** message from other node except parent, it sends its acknowledgement.
- The build tree phase is shown in Fig. 3.3.7. Node P broadcast **Election** message to Q and Y. Node Q broadcast **Election** message to R and V. In this scenario, Y is slow in sending message to V. As a result, V receives message fast from Q compared to from Y. Node V broadcast message to T and W and node R broadcast to S and T. After this, node T receives the broadcast message from V before R. Node W forward the message to T and X and further, T forward message to node U which gets the message faster from T and S. This process continues till traversing of all the nodes is finished.

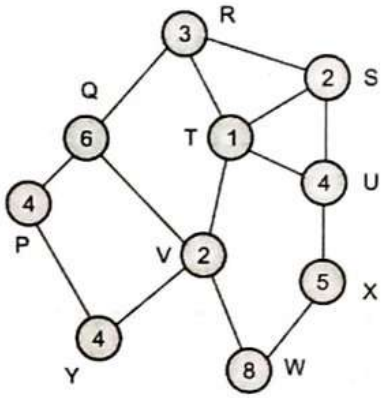


Fig. 3.3.7

- In this method of forwarding the message, each process records the sender node of message and the node to which message is forwarded. Receiving node of the message prepares message in order to reply in the form [node, capacity] and send it to the node from which message was received. In following Fig. 3.3.8, reporting of best node to source is shown.
- The first reply node is generated from the last node and same return path is traversed. The receiver of replied message selects the message of highest capacity if it receives multiple messages. This message it then forwards to next node along the return path.
- As shown in Fig. 3.3.8, The reply message [U,4] would be initiated from U to T and [X,5] from X to W. Node T forward the message as it is [U,4] to V. During the same time, W receives the message [X,5] and as its capacity greater than that of X, it replaces the message to [W,8] and forward it to V.
- As stated previously, node V selects the higher capacity message [W,8] and forward it to node Q. In the meantime, S initiate the reply message [S,2] to node R which compares its own capacity and forward the new message [R,3] to node Q. The node Q now receives two messages. One is [R,3] from R and [W,8] from V. The best node value [W,8] is sent to source node P. AT the same time, P also receives message [Y,4] from Y. It then selects the higher value out of these as [W,8] and reports W as a best node. Finally W is elected as new coordinator.

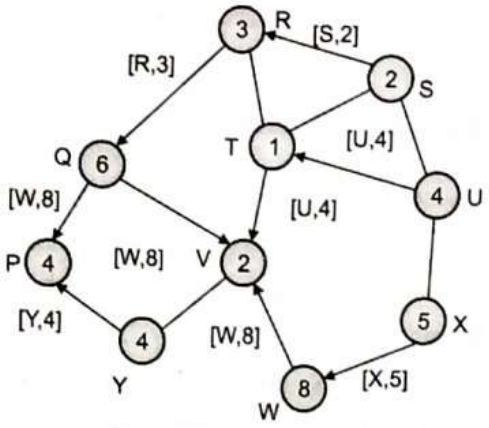


Fig. 3.3.8

3.4 Mutual Exclusion

- In environment where multiple processes are running, processes shares resources. When process has to read or update shared data structures, it enters in critical section (CS) in order to achieve mutual exclusion.
- This ensures that, no other process enters the critical in order to use shared data structure when a process is already using it.

3.4.1 Distributed Mutual Exclusion

- In uniprocessor system, semaphore, monitor and other tools are used to achieve mutual exclusion. In distributed system, since processes runs of different machines and also resources are distributed on many machines, different solutions are required to achieve mutual exclusion. In distributed system shared memory may does not exist. Hence, algorithms based on message passing are used to achieve mutual exclusion in distributed system.
- In distributed system, the problem of mutual exclusion becomes much more complex as shared memory is not available and there can be problem of lack of global physical clock. The message delays can be also unpredictable. Therefore, node in the system cannot have the complete knowledge of the overall state of the system.

3.4.2 Classification of Mutual Exclusion Algorithms

- Many algorithms are developed to achieve mutual exclusion in distributed system. These algorithms differ as per topology of network and amount of information which each node maintains about other nodes. These algorithms are classified as follows :
- **Non-token based Algorithms :** These algorithms require two or more consecutive exchanges of messages between nodes. These algorithms are based on declaration, as algorithm permits process to enter in critical section on any node as declaration of local variable on that node permits it. As declaration of local variable becomes true, process enters in critical section on that node. At any node, the mutual exclusion is enforced in this class of algorithms due to true value of local variable declared.
- **Token based Algorithms :** These algorithms uses unique tokens shared among nodes in distributed system. If process on any node receives token and process it, then it can enter in critical section as long as it holds the token.

- When process finish execution in critical section. These classes of algorithms differ on the basis of the way in which nodes/processes searches the tokens.

3.4.3 Requirements of Mutual Exclusion Algorithms

- Algorithm should guarantee that only one process should be in critical section to read or update shared data. Mutual exclusion algorithms should have following features.
- **Algorithms should not involve in deadlocks :** More than two nodes in distributed system should not wait endlessly for the messages which will never be received.
- **Avoidance of Starvation :** If some nodes are repetitively executing critical section and some any other node is indefinitely waiting for the execution of critical section, such situation should be avoided. In finite time, every requesting node should get opportunity to execute in critical section (CS).
- **Fairness :** As there is absence of physical global clock, either requests at any node should be executed in the order of arrival or in the order these were issued. This is due to consideration of logical clocks.
- **Fault Tolerance :** In case of any failure in the course of carrying the work, if any failure occurs, then mutual exclusion algorithms should reorganize itself to carry out the further designated job.

3.4.4 Performance Measure of Mutual Exclusion Algorithms

Following four metrics measure the performance of mutual exclusion algorithms in distributed system.

1. The number of required messages for invocation of critical section in first attempt.
2. The delay incurred between the moments when one node leaves the CS and before next node enters the CS. This time is also called as synchronization delay. In this case, more messages exchange may require during this delay duration.
3. First node's CS request arrives. Then its request message is sent out to enter in CS. Response time is the time interval between the request message sent out to enter in CS and when node exit the CS. It is shown in Fig. 3.4.1.
4. The rate at which system executes request that arrives for critical section (CS) which is called as system throughput.

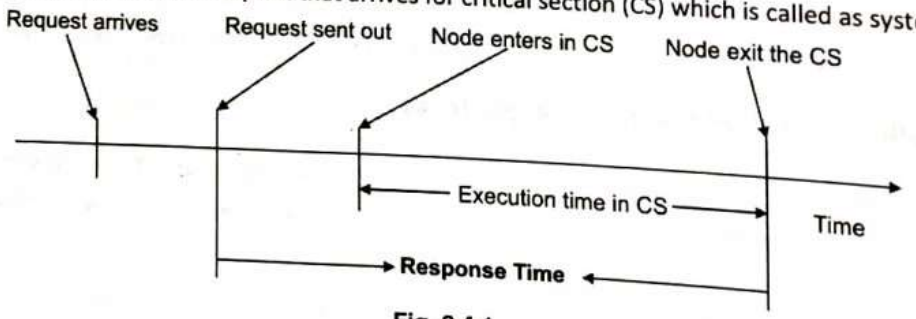


Fig. 3.4.1

3.4.5 Performance in Low and High Load Conditions

In low load condition of the system, there is rarely more than one request at the same time in system. In high load condition of the system, there is often pending request for mutual exclusion at the node (site). A node rarely remains idle in high load condition of the system. In best case performance, performance metrics achieves best possible values.

3.5 Non-Token Based Algorithms

- In non-token based mutual exclusion algorithms, a particular node communicates with set of nodes to decide who should enter in critical section next. Requests to enter in CS are ordered on the basis of timestamps in non-token based mutual exclusion algorithms.
- Simultaneous requests are also handled by using timestamp of the request. Lamport's logical clock is used and request for CS having less timestamp value gets priority over requests having larger timestamp values.

3.5.1 Lamport's Algorithm

Request for CS

- In this algorithm, for all $i : 1 \leq i \leq N :: R_i = \{S_1, S_2, \ldots, S_N\}$, that is there are N number of sites. Single process run on each site. Site S_i represents process P_i. The **request_queue$_i$** , is maintained by every site i to store mutual exclusion requests in the order of timestamps. It is necessary to deliver messages in FIFO order between each pair of sites.
- Whenever site S_i want to enter in critical section (CS), it sends a message **REQUEST(ts_i, i)** to all the sites in its request set R_i and puts the message in **request_queue$_i$**. The timestamp of the request is (ts_i, i).
- The site S_j sends timestamped **REPLY** message to Si after receiving **REQUEST(ts_i, i)** message from S_i. This request of S_i, site S_j puts in its **request_queue$_j$**.

Executing the CS

- Site Si can enter in CS if following two conditions are satisfied.
- **C1 :** Site S_i has received message from all the sites with timestamp larger than (ts_i, i).
- **C2 :** S_i's request is at the top of **request_queue$_i$**.

Exiting the CS

- When S_i exit the CS, it removes the request from top of **request_queue$_i$**. S_i then sends timestamped **RELEASE** message to all the sites in its request set.
- After receiving **RELEASE** message from S_i, site S_j removes S_i's request from its **request_queue$_j$**.

Correctness

- Lamport's algorithm ensures that, only one process will be execution in CS at any time. If we assume both S_i and S_j executing in S then both have their own requests at the top of their request queues and C1 holds.
- If request of S_i has lesser timestamp than request of S_j ($T(S_i) < T(S_j)$), then as per C1 and FIFO delivery of channels, request of S_i should present in **request_queue$_j$**. If S_j is executing then its request is at top of **request_queue$_j$**. It is not possible as $T(S_i) < T(S_j)$. Hence algorithm achieves mutual exclusion.

3.5.2 Ricart–Agrawala's Algorithm

- Ricart–Agrawala's algorithm is an optimization of Lamport's algorithm. The working of the algorithm is as follows :
- When process wants to enter in CS, it sends the **REQUEST** message to all other processes. This **REQUEST** message contains the name of CS in which it (sender process) wants to enter, its process number and the current time. This message also conceptually delivers to the sender. Each message is acknowledged by the receiver.
- The receiver of the message takes action as per its current state with respect to name of the CS mentioned in the received message. Following three cases occur.
  1. Currently if receiver is not in CS and also does not want to enter it then it replies **OK** message to sender.
  2. If receiver is already running in CS then it does not reply. It puts the message in its queue.
  3. If receiver is about to enter in CS and yet has not done then it compares the timestamp in incoming request message with the timestamp in request message which it has already sent to other processes. The lowest one wins. Receiver sends OK message to the sender if incoming request message has lower timestamp value. If its own request message has lower timestamp value then it puts the incoming request message in its queue and does not send any reply.
- When requesting process for CS receives reply as **OK** messages from all the other processes, it enters in CS. If process is already in CS then it replies OK message to other requesting process after coming out of CS.

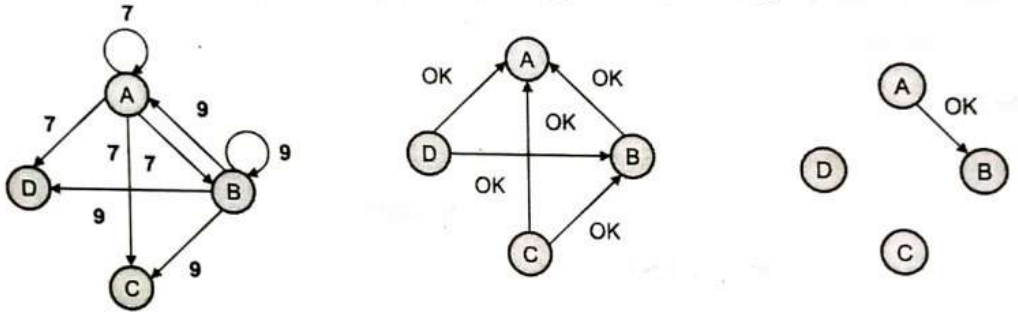


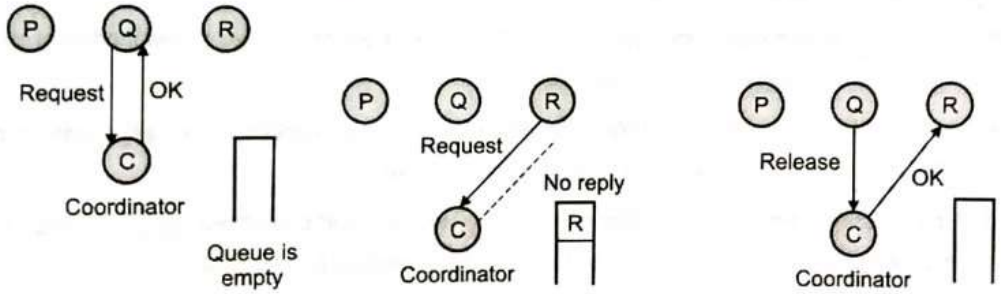
Fig. 3.5.1

- As shown in Fig. 3.5.1 (a), process A and B both wants to enter in CS simultaneously. Process A and B both sends **REQUEST** message to everyone with timestamp 7 and 9 respectively.
- As shown in Fig. 3.5.1 (b), as process C and D both are not interested in entering the CS, both sends OK message as reply to sender processes A and B. Process A and B both see conflict and compare timestamps. In this case process A wins as its message timestamp 7 is lower than the timestamp of message sent by process B, which is 9.
- The process A enters in CS and keeps request of process B in its queue. When process A exits the critical section, it sends OK message to process B so that later process B will enter in CS. It is shown in Fig. 3.5.1 (c). Hence, algorithm ensures the mutual exclusion in distributed system as lower timestamp process is allowed to enter in CS in conflict situation.

- If we consider n number of processes in the system, then process sends (n–1) REQUEST messages to enter in CS. It receives (n–1) OK messages from other processes to get permission to enter in CS. Hence, 2(n–1) messages are required to exchange to enter in CS. There is no single point of failure.
- In this algorithm, if reply or request if lost then sender gets blocked due to time outs. Sender of request message then may conclude that receiver is dead or it may try continuously sending the request. Sender has to wait for **OK** message from destination.
- For large size group of processes, each process should maintain list of members. Group members may leave the group, new members may join the group or group member may crash. Hence, algorithm produces best performance and success for small size groups. This algorithm is complex, slower, expensive and less robust.

3.5.3 Centralized Algorithm

- In this algorithm, a central coordinator process is responsible to give permission to other processes to enter in critical section (CS). Coordinator process keeps track of which CS is busy or currently in execution. Process which wants to enter in CS sends **REQUEST** message to coordinator stating the name of CS.
- If CS is free then coordinator sends **OK** message to requesting process to grant permission to enter in stated CS in message. If in stated CS, already other process is executing then it puts the request message in its queue.
- When currently executing process in CS exits, it sends **RELEASE** message to coordinator process. As a result, coordinator knows that CS is now free and it takes request from the queue to grant permission to enter in CS. If requesting process is still blocked, it unblocks and enters in CS.



(a) (b) (c)

Fig. 3.5.2

- As shown in Fig. 3.5.2 (a), Process C is coordinator and currently no process is executing in CS. Process Q sends **REQUEST** message to coordinator and it permits process Q to enter in CS by replying with **OK** message.
- As shown in Fig. 3.5.2 (b), process R also send **REQUEST** message to coordinator to get permission to enter in same CS. As coordinator knows process R is already in CS, it puts R's request in its queue.
- As shown in Fig. 3.5.2 (c), when process Q exit the CS it sends **RELEASE** message to coordinator. Now coordinator knows that CS is free, it takes message from queue and grants permission to process R by replying with **OK** message. Now process R enters in CS.

- This algorithm guarantees mutual exclusion as at a time one process is allowed to enter in CS. This algorithm requires 3 messages to enter and exit the CS (Request, OK and Release). Limitation of the algorithm is that coordinator may crash.

3.5.4 Maekawa's Algorithm

- In Ricart–Agrawala's algorithm, process should get permission from all other processes to enter in CS. In Maekawa's algorithm, to enter in CS, process has to obtain permission from subset of the processes as long as subsets used by any two processes overlaps. Process need not obtain permission from all the processes in order to enter in CS.
- Processes carry out voting to one another in order to enter in CS. If process collects enough votes, then it can enter in CS. The conflict is resolved by the processes common to both sets of processes. Hence, processes in intersection of two sets of voters guarantee the safety property that; only one process enters in CS, by voting to a single candidate to enter in CS.
- For all the processes 1 to N, there is associated voting set V_i with process P_i. The set V_i is subset of all the processes numbered from 1 to N (P_1, P_2, P_3 ..... P_N). For all processes i and j between 1 to N, this set satisfies following four properties.
- Process P_i is member of V_i.
  1. Voting sets V_i and V_j contains minimum one common member.
  2. Size of voting set is same. Let it be K.
  3. Each process P_j belongs to X of the voting set V_i.
- In this algorithm, optimal solution is provided to minimize the value of voting set size K specified in property 3. This solution guarantees mutual exclusion for K - √ N and X equal to K.
- In order to get access to CS, process P_i sends **REQUEST** message to K–1 members of V_i. Process P_i can enter is CS if it receives reply from K–1 processes of the set. Let Process P_j belongs to set V_i.
- Process P_j sends reply message immediately to process P_i after receiving the **REQUEST** message from it unless either its state is **HELD** or it has replied (**VOTED** message) since it last received the **RELEASE** message.
- When process receives **RELEASE** message, it takes out the outstanding **REQUEST** messages from front of the non-empty queue and sends reply as **VOTE** message. To exit the CS, P_i sends **RELEASE** messages to all (K–1) members of V_i.
- This algorithm guarantees mutual exclusion as common processes in V_i and V_j should cast votes for both processes P_i and P_j which simultaneously want to enter in CS. But this algorithm permits to cast at the most one vote between consecutive receipts of **RELEASE** message. This is impossible and hence, ensures mutual exclusion.
- This algorithm leads to deadlock situation. Let $V_1 = \{P_1, P_2\}$, $V_2 = \{P_2, P_3\}$ and $V_3 = \{P_3, P_1\}$. If all three processes P_1, P_2 and P_3 simultaneously want to enter in CS then P_1 may reply to P_2 while holding P_3. Process P_2 to reply P_3, it may hold P_1. Process P_3 to reply P_1, it may hold P_2. As in this situation each process will receive one out of two reply, none can proceed further.
- In this algorithm, if process crashes and it is not the member of voting set then its failure does not affects other processes. The steps of the algorithm are summarized as shown below.
  1. Initially state is **RELESED** and **VOTED** is set as false.

2. If P_i wants to enter in CS then its state is **WANTED** and it sends **REQUEST** messages to all the processes $(V_i - \{P_i\})$ in its set of voting size K. It waits till arrival of (K-1) reply. State = **HELD**.
3. If P_j receives reply from P_i $(i \neq j)$ and if state = **HELD** or **VOTED** = true then puts the **REQUEST** of P_i in queue and don't reply. Otherwise reply to P_i and **VOTED** = true
4. For P_i to exit the CS, state = **RELESED** and reply **RELESE** message to all processes $V_i - \{P_i\}$.
5. When process P_j receives **RELESE** from P_i where $(i \neq j)$ and if queue of request have messages (non-empty) then remove message from front (head) of queue. Suppose it is from P_k. Then send reply to P_k. Now **VOTED** = true. Otherwise **VOTED** = false.

3.6 Token Based Algorithms

- Token based algorithm makes use of token and this unique token is shared among all the processes. The process possessing token can enter in CS. There are different token based algorithms based on the way process carry out search for the token.
- These algorithm uses sequence numbers instead of timestamps. Each request for the token has sequence number and these sequence numbers for processes advances independently. Whenever process request for the token, its sequence number gets incremented. This helps is differentiating old and current request for the token. As only process acquiring token enters in CS, algorithms ensure mutual exclusion.

3.6.1 Suzuki-Kasami's Broadcast Algorithm

- In this algorithm, suppose process want to enter in CS and does not holds token then it broadcast a **REQUEST** message to all the processes. After receiving the **REQUEST** message, process holding token sends it to requesting process.
- If process is already in CS when receives the **REQUEST** message then it sends token only after it has exited the CS. Process holding token can repeatedly enter in CS till it has a token. Following two issues are important to consider in this algorithm.
  - To differentiate outdated **REQUEST** messages from current **REQUEST** messages.
  - Determining the process having outdated request for CS.
- Let **REQUEST(P_j, n)** is the **REQUEST** message of process P_j. Here, n (n = 1, 2, 3...) is a sequence number that shows P_j is requesting its n^{th} CS execution. Process P_i maintains array of integers $RN_i[1...N]$ where $RN_i[J]$ indicates largest sequence number received from P_j. If process P_i receives **REQUEST(P_j, n)** then this message will be outdated if $RN_i[J] > n$. When process P_i receives **REQUEST(P_j, n)**, it sets $RN_i[J] = \max(RN_i[J], n)$.
- Token has queue of requesting processes and array of integers SN[1...N]. Where SN[j] indicates the sequence number of the request that has executed by process P_j most recently. Process P_i sets $SN[i] = RN_i[i]$ whenever it finishes its execution of CS. It shows that its request corresponding to sequence number $RN_i[i]$ has been executed.
- At process P_i, if $RN_i[j] = SN[j] + 1$ then process P_j is currently requesting the token. The process which has executed the CS now checks this condition for all j's to know about all the processes that are requesting the token and puts their IDs in requesting queue if not already there. Process then sends token to the process at head of requesting queue.
- This algorithm is simple and efficient. It requires 0 or N messages per CS entry. Synchronization delay is 0 or T. If process has idle token while requesting then no message or zero delay is required.

Algorithm

Request for CS

(i) If process P_i wants to enter in CS and does not have token, then it increments its sequence number $RN_i[i]$. It then sends **REQUEST(P_i, sn)** message to all the processes. In this message sn is updated value of $RN_i[i]$.

(ii) After receiving this request message, process P_j sets $RN_j[i]$ to max($RN_j[i]$, sn). If P_j has token then it sends to P_i if $RN_j[i] = SN[i] + 1$.

Executing the CS

After receiving the token, process P_i executes in CS.

Exiting the CS

Process P_i carry out following updates after it finishes execution of CS.

(i) It sets $SN[i] = RN_i[i]$.

(ii) For all processes P_j, if their IDs does not belong to the token queue then append these IDs in it if $RN_i[j] = SN[j] + 1$.

(iii) If token queue is still has some messages in it then delete ID at head of queue and sends token to the process of that ID.

3.6.2 Singhal's Heuristic Algorithm

- In this algorithm, each process keeps information about states of other processes in the system. This information is used by the process to determine the set of processes that likely to have the token. The process then requests the token from these processes to reduce the number of messages needed to enter in CS.
- The request for token should go to the process which is holding the token or going to obtain it in near future. Otherwise there can be chances of deadlock or starvation.
- Process P_i keeps two arrays $PV_i[1..N]$ and $PN_i[1..N]$ to store the state and highest known sequence number of each process respectively. Token also maintains two arrays TPV[1..N] and TPN[1..N]. Outdated requests are determined by using sequence numbers. Process can be in one of the following state.
  - **REQ** : Requesting the CS.
  - **EXE** : Executing the CS
  - **HOLD** : Holding the idle token.
  - **NONE** : None of above.

Initially arrays are set as follows

- For every process P_i, for i = 1 to N do
  - Set $PV_i[j]$ = **NONE** for j = N to i
  - Set $PV_i[j]$ = **REQ** for j = i – 1 to 1
  - Set $PN_i[j] = 0$ for j = 1 to N.
- Initially process P_1 is in **HOLD** state. Hence, $S_1[1]$ = **HOLD**.
- For the token TPV[j] = **NONE** and TPN[j] = 0 for j = 1 to N.

- Above initialization ensures that either $PV_i[j]$ = **REQ** or $PV_j[i]$ = **REQ.** Hence, for any two processes sending request at the same time, one process will always send token request to other. This guarantees that processes are not inaccessible from each other and REQUEST message from process will be delivered to process holding the token or to the process which will hold it in near coming time.

Algorithm

Request for CS

1.If process P_i wants to enter in CS and does not have token, then it takes following action.

- Sets $PV_i[i]$ = **REQ.**
- Sets $PN_i[i] = PN_i[i] + 1$.
- P_i then sends **REQUEST(P_i, sn)** message to all the processes P_j for which $PV_i[j]$ = **REQ.** In this message **sn** is updated value of $PN_i[i]$

2.After receiving the message **REQUEST(P_i, sn)** by P_j, it discards this message if $PN_j[i] \geq sn$ as message is outdated now. Otherwise, it (P_j) sets $PN_j[j]$ = sn and carry out following activities as per its current state.

- $PV_j[j]$ = **NONE** : Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **REQ** : If $PV_j[i] \neq$ **REQ** then set $PV_j[i]$ = **REQ** and send a **REQUEST(P_j, $PN_j[j]$)** message to P_i (Else do nothing).
- $PV_j[j]$ = **EXE** : Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **HOLD** : Set $PV_j[i]$ = **REQ,** TPV[i] = **REQ,** TPN[i] = sn, $PV_j[j]$ = **NONE** and send the token to process P_i.
- $PV_j[j]$ = **NONE** : Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **REQ** : If $PV_j[i] \neq$ **REQ** then set $PV_j[i]$ = **REQ** and send a **REQUEST(P_j, $PN_j[j]$)** message to P_i (Else do nothing).
- $PV_j[j]$ = **EXE** : Set $PV_j[i]$ = **REQ.**
- $PV_j[j]$ = **HOLD** : Set $PV_j[i]$ = **REQ,** TPV[i] = **REQ,** TPN[i] = sn, $PV_j[j]$ = **NONE** and send the token to process P_i.

Executing the CS

3.Process P_i executes CS after receiving the token. Prior to entering in CS it sets $PV_i[i]$ = **EXE.**

Exiting the CS

4. After finishing the execution of CS, process P_i sets $PV_i[i]$ = **NONE** and TPV[i] = **NONE.** After this, it updates its array as mentioned below.

```
For all Pj for j = 1 ....N do
        If PNi[j] > TPN[j]
                Then
                        // Use token information to update local information.
                        {TPV[j] = PVi[j]; TPN[j] = PNi[j]}

                Else
                        // Use local information to update token information.
                        {PVi[j] = TPV[j]; TNi[j] = TPN[j]}
```

5. If for all j, $PV_i[j]$ = **NONE** then set $PV_i[i]$ = **HOLD,** otherwise send token to process P_j such that $PV_i[j]$ = **REQ.**

- Fairness of algorithm will rely on selection of process P_i, as no queue is available in token. In this algorithm, negotiation rules are used in order to ensure fairness. Algorithm requires average N/2 messages in low to moderate load condition. During high load, when all processes requests CS, N number of messages are required. Synchronization delay required is T.

Example

Assume there are 3 processes in the system. Initially:

- Process 1 : $PV_1[1]$ = **HOLD**, $PV_1[2]$ = **NONE**, $PV_1[3]$ = **NONE**. $PN_1[1]$, $PN_1[2]$, $PN_1[3]$ are 0.
- Process 2 : $PV_2[1]$ = **REQ**, $PV_2[2]$ = **NONE**, $PV_2[3]$ = **NONE**. PNs are 0.
- Process 3 : $PV_3[1]$ = **REQ**, $PV_3[2]$ = **REQ**, $PV_3[3]$ = **NONE**. PNs are 0.
- Token : TPVs are NONE. TSNs are 0.

Assume P_2 is requesting token.

- P_2 sets $PV_2[2]$ = **REQ**, $PN_2[2] = 1$.
- P_2 sends **REQUEST(P_2,1)** to P_1 (As only P_1 is set to **REQ** in PV[2])
- P_1 receives the **REQUEST**. Accepts the **REQUEST** since $PN_1[2]$ is smaller than the message sequence number.
- As $PV_1[1]$ is **HOLD**: $PV_1[2]$ = **REQ**, TSV[2] = **REQ**, TSN[2] = 1, $PV_1[1]$ = **NONE**.
- Send token to process P_2.
- P_2 receives the token. $PV_2[2]$ = **EXE**. After exiting the CS, $PV_2[2]$ = TPV[2] = **NONE**.
- Updates PN, PV, TPN, TPV. Since nobody is requesting, $PV_2[2]$ = **HOLD**.
- Suppose P_3 issues a **REQUEST** now. It will be sent to both process P_1 and P_2. Only P_2 responds since only $PV_2[2]$ is **HOLD** ($PV_1[1]$ is **NONE** at the present).

3.6.3 Raymond's Tree-Based Algorithm

- In this algorithm, processes are organized in directed tree. In this tree, edges of the tree are towards the root (process) which currently having token. Each process in tree maintains local variable **HOLDER (H)** that points to an immediate neighbor on path towards root process. The local variable **HOLDER (H)** can also point to the root. The process which currently holds the token has privilege to be root.
- If we chase **HOLDER (H)** variables, every process has path towards process holding token. At root, **HOLDER (H)** points to itself. Every process maintains FIFO **Request-queue** with it which stores requests sent by neighboring processes but has not so far sent the token. Following are the steps in algorithm.
- Consider two processes P_i and P_j where P_j is neighbor of P_i. If **HOLDER** variable $H_i = j$. It means P_i has information that, root can be reached through P_j. In this case, undirected edge between P_i and P_j converts to directed edge from P_i and P_j. Hence, H variable helps to trace path towards process holding the token.

- Consider undirected graph with processes P_1 to P_7 with holder variable H. Process P_1currently hold the token. P_1 and P_2 are the neighbors of root P_1; P_4 and P_5 of P_2 and processes P_6 and P_7 of P_3. Every process keeps information about its neighbor and communicates with only to its immediate neighboring processes which lead to root process P_1.
- Suppose P_4 wants to enter in CS. Process P_4 sends **REQUEST** message to process P_2 as $H_4 = 2$. Process P_2 in turn forward this message to process P_1 ($H_2 = 1$). This **REQUEST** message gets forwarded by the processes along the minimal spanning path on the behalf of the process which initially sends **REQUEST** message to root having token. The Forwarded REQUEST message is noted by using variable "asked" in order to prevent the resending of the same message.

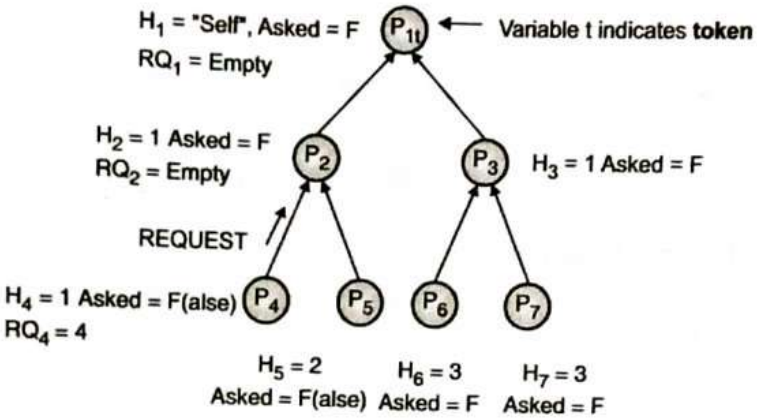


Fig. 3.6.1 : Process P_4 sends REQUEST message to P_2 which forwards it to process P_1

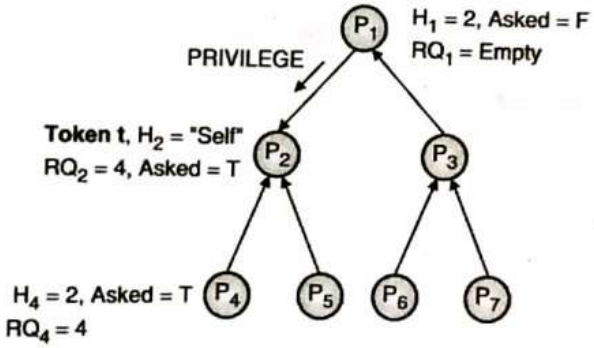


Fig. 3.6.2 : Process P_2 receives token after granting the privilege from process P_1

- Initially token was with process P_1, When it receives REQUEST message (H_1 = "Self") and if not executing CS, it sends PRIVILEGE message to its neighbor P_2. Process P_2 was the sender of REQUEST message to P_1. Now P_1 resets holder variable $H_1 = 2$. Process P_2 in turn forwards PRIVILEGE message to process P_4. Process P_2 had requested token on the behalf of process P_4, it updates holder variable as $H_2 = 4$.

Data Structures

- RQ_i is the request queue maintained by process P_i ($i = 1, 2 \ldots$). It holds the identities of immediate neighbor processes that have requested for the privilege but have not yet been sent the token. Maximum size of RQ is number of immediate neighbors + 1 (process itself).

– Each process has variable "Asked" which is initially set as false (F). Its value becomes true (T) only if REQUEST is sent either for itself or neighbors and not received. After receiving the PREVILEGE, its value is again set to false.

Algorithm

1. Request for CS

(If token is already held by the process then no need to send REQUEST message. Process needs token for itself or for its neighbor. If variable Asked is false, means it has not already sent a request message).

If process P_i does not hold the token and RQ_i is not empty and Asked = F

Then

RQi = RQ_i + 1 and send message to the process held in holder variable H_i;

Asked = T;

Else Asked = F;

2. P_j Receives REQUEST message from P_i.

If P_j holds token ant if it is not executing CS and RQ_j is not empty and at the head of its RQ ID is not "self"

Then

Send PRIVILEGE message to requesting process P_i; update H_j = i;

Else if process is unprivileged process

Process forwards the message to parent process.

3. Receipt of PRIVILEGE message

If P_i had requested the token; H_i = "Self"

Then

RQi = RQ_i – 1; Enter in CS; Asked = F;

If P_i had forwarded the request on the behalf of its neighbor;

Then

RQj = RQ_i – 1; Send the PRIVILEGE message to requesting process which is top of RQ_i; update H_i and change parent

If RQi is still not empty

Then send request message to new H value; Asked = T;

Else Asked = F

4. Execution of CS

Process Pi can enter the CS if it has token and its own ID is at the head of RQ;

5. Process P_i Exiting the CS

If RQi is not empty

Then

Dequeue RQi; If this element is ID of P_n; Send the token to P_n; $H_j = n$; Asked = F;

If RQi is still not empty

Then send REQUEST to parent process; Asked = F.

Performance

Algorithm exchanges 4 messages if system load is heavy. It exchanges only O(log N) messages under light node.

3.6.4 Token Ring Algorithm

- In this algorithm, processes are logically arranged in ring in bus based network. There is no any fix criterion to arrange the processes in ring. IP addresses of the machines can be used or other criterion can be used. Each process should know the next process in line after itself.
- Initially process P_0 holds the token. Token passes from process k to process k+1 in point to point messages. After acquiring token from its neighbor, process checks if it has to enter in CS. If yes, then it enters the CS. After it finishes its execution of CS and exit the CS, it passes the token to its successor along the ring. Same token cannot be used to enter second CS.
- In this way token circulates in ring. If no process is interested to enter in CS then simply token circulates with high speed in ring. As only one process has the token at any instant, a single process will be there in CS at a time.
- It ensures mutual exclusion. There is no starvation. The problem in this algorithm is loss of token. After lost, token needs to be regenerated. If successor of the process is dead to which token is passed then it is removed from ring. Messages required per entry/exit are 0 to infinity. Delay before entry is 0 to (n–1) messages.

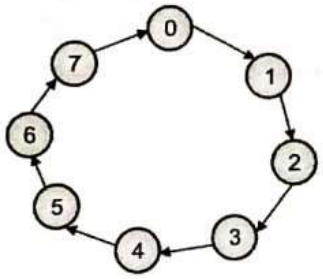


Fig. 3.6.3

3.7 Comparative Performance Analysis of Algorithms

Three parameters are used to compare performance of the algorithms. These are response time, Number of messages required and synchronization delay.

3.7.1 Response Time

- In low load condition of system, response time for many algorithms is round trip time to acquire token *(2T)* plus the time required to execute the CS (which is *E*). In Raymond's Tree-Based algorithm, average distance between requesting process and process holding token is *(log N)/2*. Average round trip delay therefore is *T(log N)*.

- Different algorithms vary in their response time in high load condition of the system. It varies when load in system increases.

3.7.2 Synchronization Delay

- This is delay due exchanges of messages needed after process exit the CS and before next site enters the CS. In most of the algorithms when process exits the CS, it directly sends REPLY or token message to the next process to enter the CS. As a result, the synchronization delay required is T.
- In Maekawa's Algorithm, process exiting the CS unlocks arbiter process by sending RELEASE message. After this, arbiter process sends GRANT message to next process to enter the CS. Hence, synchronization delay required is 2T.
- In Raymond's algorithm, two processes that consecutively execute the CS can be found at any position in tree. Token passes serially along the edges in tree to the process to enter CS next. If tree has N nodes then average distance between two nodes is (log N)/2. The synchronization delay required is T(log N)/2.

3.7.3 Message Traffic

- Lamport's, Ricart-Agrawala, and Suzuki-Kasmi algorithms respectively needs 3(N-1), 2(N-1) and N messages per CS execution in any load condition. In other algorithms, messages required depends on whether light or heavy load present in the system.
- **Light Load :** Raymond's algorithm requires log N messages per CS execution. The request message travel upward towards root node from requester. The token travel backs from root to requester node. If tree has N nodes then average distance between root node and requester node nodes is (log N)/2. Singhal's heuristic algorithm requires N/2 messages per Cs execution.
- **Heavy Load :** Maekawa's Algorithm requires 5√ N messages in high load condition. Raymond's algorithm requires 4 messages whereas, Singhal's heuristic algorithm requires N messages.
- Following table gives comparative performance of different algorithms.

Algorithm	Response Time (Light Load)	Synchronization Delay	Messages (Light Load)	Messages (High Load)
Lamport (Non-Token)	2T+E	T	3(N–1)	3(N–1)
Ricart Agrawala (Non-Token)	2T+E	T	2(N–1)	2(N–1)
Maekawa (Non-Token)	2T+E	2T	3√ N	5√ N
Suzuki and Kasmi (Token)	2T+E	T	N	N
Singhal's Heuristics (Token)	2T+E	T	N/2	N
Raymond (Token)	T(log N)+E	T	Log(N)	4

Review Questions

Q. 1 Why clock synchronization is required in distributed system? Explain.

Q. 2 Explain physical clocks in detail.

Q. 3 Write short note on Global Positioning System (GPS).

Q. 4 Explain Cristian's algorithm for clock synchronization.

Q. 5 Explain Berkeley algorithm for clock synchronization.

Q. 6 Explain Averaging algorithms for clock synchronization.

Q. 7 Explain how Clock Synchronization is carried out in Wireless Networks.

Q. 8 Explain working of Network Time Protocol (NTP).

Q. 9 What is logical clocks? Explain Lamport's logical clock with example.

Q. 10 What is total order multicasting? Explain with example.

Q. 11 What is vector clocks? Explain.

Q. 12 Explain Bully election algorithm with example.

Q. 13 Explain Ring election algorithm with example.

Q. 14 How election of coordinator is carried out in wireless networks? Explain.

Q. 15 What is mutual exclusion? How it is different in distributed system?

Q. 16 Explain in short working ofNon-token based and token-based algorithms for mutual exclusion.

Q. 17 What are the requirements of mutual exclusion algorithms.

Q. 18 Explain metrics to measure performance of mutual exclusion algorithms.

Q. 19 Explain Lamport's algorithm for mutual exclusion in distributed system.

Q. 20 Explain Ricart–Agrawala's algorithm for mutual exclusion in distributed system.

Q. 21 Explain centralized algorithm for mutual exclusion in distributed system.

Q. 22 Explain Maekawa's algorithm for mutual exclusion in distributed system.

Q. 23 Explain Suzuki-Kasami's Broadcast algorithm for mutual exclusion in distributed system.

Q. 24 Explain Singhal's Heuristics algorithm for mutual exclusion in distributed system.

Q. 25 Explain Raymond's Tree-Based algorithm for mutual exclusion in distributed system.

Q. 26 Explain performance analysis of different mutual exclusion algorithms.

□□□

Resource and Process Management

Syllabus

Desirable Features of global Scheduling algorithm, Task assignment approach, Load balancing approach, load sharing approach, Introduction to process management, process migration, Threads, Virtualization, Clients, Servers, Code Migration.

4.1 Introduction

- In distributed system, resources are distributed on many machines. If local node of the process does not have needed resources then this process can be migrated to another node where these resources are available.
- In this chapter, resource is considered as processor of the system. Each processor forms the node of the distributed system.
- Following techniques are used for scheduling the processes of the distributed system
  - **Task assignment approach :** In this method, tasks of each process are scheduled to suitable nodes to improve performance.
  - **Load-balancing approach :** In this approach, main objective is to distribute the processes to different nodes to make workload equal among the nodes in the system.
  - **Load-sharing approach :** In this approach, main objective is to guarantee that no node will be idle while processes in system are waiting for processors.

4.2 Desirable Features of Global Scheduling Algorithm

Following are the desirable features of good global scheduling algorithm

- **No Priori Knowledge about the Processes :** Process scheduling algorithm should not consider a priori knowledge about the characteristics and resources need of the processes. It imposes extra burden on users as they should specify this information while submitting the job for execution.
- **Dynamic in Nature :** Process scheduling algorithm should always consider the current information about load of the system and should not obey any fixed static policy to assign the process to particular node. It should always consider dynamically changing load on the nodes of the system. Hence, algorithm should have flexibility to migrate a process more than once. This is necessary as after some time system load may change. Hence, it is necessary to transfer the process from previous node to new node again to adapt to the changed system load. This is possible only if the system support preemptive process t
- **Quick Decision Making Capability :** Process scheduling algorithm should always make quick decisions about regarding assigning the processes to nodes.
- **Balanced System Performance and Scheduling Overhead :** Process scheduling algorithm should not collect more global state information in order to use it for process assignment decisions.

- In distributed system, collecting global state information requires more overhead. Further, due to aging of information being collected and low scheduling frequency due to cost of collecting and processing, usefulness of information can reduce. Hence, algorithm offering near optimal performance by using minimum global state information is preferred.
- **Stability :** There may be a situation where all the nodes in the system spends their time maximum time in migrating the processes without carrying out any useful work. This migration of processes is carried out to schedule the processes for achieving the better performance. Such unproductive migration is called processor thrashing. Processor thrashing occurs if nodes are capable of scheduling its own processes locally and independently. In this case, node takes scheduling decision based on local information and does not coordinate with other nodes for the same. It may also happen that, the information at node may become old due to transmission delay between the nodes. Suppose node 1 and node 2 observes that node n3 is idle, both sends their part of the load to node 3 without coordinating with each other. In this situation, if node 3 becomes overloaded then it also starts transferring its processes on other nodes. This leads to unstable state of the system due to repetition of such cycle again and again.
- **Scalability :** Process scheduling algorithm should work for small as well as large networks. If algorithm enquires for lightly loaded node and sends processes there then it can work better for smaller networks. In large network, as many replies needs to be processes it consumes bandwidth as well as processing power. Hence, it cannot be scalable. Better to probe k out of n nodes.
- **Fault Tolerance :** Algorithm should continue its working although one or more nodes in system crashes. It should consider available nodes in decision making.

4.3 Task Assignment Approach

- This approach finds an optimal assignment policy for the tasks of the individual process. Following are the assumptions in this approach.
  - Tasks of the process are less interdependent and data transfer among them will be minimized.
  - How much computation is required for task and speed of processor is already known.
  - Cost of processing needed by each task on every node is known.
  - Intercrosses communication (IPC) cost between each pair of tasks is known and it is zero if communicating tasks are running on the same node.
  - Precedence relationship among tasks, resources need of each task, resources available on each node is already known.
- Considering above assumptions, the main goal of assigning the tasks of the process to nodes in the manner to achieve following.
  - Reducing IPC cost
  - Quick turnaround time for complete process
  - A high degree of parallelism
  - Efficient utilization of system resources.
- It is not possible to achieve all these goals simultaneously as they conflicts each other. You can minimize IPC by keeping all tasks on same node. But for efficient utilization of resources, tasks must be evenly distributed among nodes. Precedence relationship among tasks also limits their parallel execution. If x number of tasks and y number of nodes then x^y are the possible assignments of tasks to nodes. In practice, certain tasks cannot be assigned to certain nodes due to some restrictions. Hence, possible assignments can be less than x^y.

- The total cost which is optimal not necessarily will be gained by assigning equal no of processes to each node. For two processors, optimal assignment can be done in polynomial time. But for arbitrary number of processors the cost is NP hard. Heuristics algorithm found to be efficient from computation point of view but produce sub optimal assignments.

4.4 Load-Balancing Approach

- Load balancing algorithms use this approach. It assumes that, equal load should be distributed among all nodes to ensure better utilization of resources. These algorithms transfers load from heavily loaded nodes to lightly loaded nodes in transparent way to balance the load in system. This is carried out to achieve good performance relative to parameters used to measure system performance.
- From user point of view, performance metric often considered is response time. From resource point of view, performance metric often considered is total system throughput. Throughput metric involves treating all users in the system fairly with making progress as well. As a result, all load balancing algorithms focuses on maximizing the system throughput.

4.4.1 Classification of Load Balancing Algorithms

- Basically, load balancing algorithms are mainly classified into two types as **static** and **dynamic** algorithms.
- Static algorithms are classified as **deterministic** and **probabilistic**.
- Dynamic algorithms are classified as **centralized** and **distributed** algorithms.
- Further distributed algorithms are classified as **cooperative** and **non-cooperative**.
- **Static** algorithms do not consider current state of system. Whereas, **dynamic** algorithms consider it. Hence, dynamic algorithms avoid giving response to system states which degrades system performance. However, dynamic algorithms need to collect information about current state of the system and they should give response about the same. As a result, the dynamic algorithms are more complex than static algorithms.
- **Deterministic** algorithms consider properties of nodes and characteristics of processes to be assigned to them to ensure optimized assignment, for example; task assignment approach. **A probabilistic algorithm uses** information such as network topology, number of nodes and processing capability of each node etc. Hence, such information helps to improve system performance.
- In **centralized** dynamic scheduling algorithm, scheduling responsibility is carried out by single node. In **distributed** dynamic scheduling algorithm, various nodes are involved in making scheduling decision, i.e. assignment of processes to processors. In **centralized** approach, system state information is collected at single node which is centralized server node. This node takes processes assignment decision based on collected system state information. All other nodes periodically send this information to this centralized server node. This is efficient approach to take scheduling decision as single node knows load on each node and number of processes needing service. Single point of failure is the limitation of this approach. Message traffic increases towards single node, hence it may become bottleneck.
- In **distributed** dynamic scheduling algorithm various nodes are involved in making scheduling decision and hence, it avoids bottleneck of collecting state information at single node. Each local controller on each node runs concurrently with others. They takes decision in coordination with other based on systemwide objective function instead of local one.
- In **cooperative** distributed scheduling algorithms, distributed entities makes scheduling decision by cooperating with each other. Whereas, **non-cooperative** distributed scheduling algorithms do not cooperate with each other for making scheduling decisions and each one acts as autonomous entity.

4.4.2 Issues in Designing Load-Balancing Algorithms

- Following are issues related to designing load-balancing algorithms.
  1. Policy for how to calculate workload of each node.
  2. Process transfer policy to decide execution of process locally or remotely.
  3. Selection of node to transfer selected process called as location policy.
  4. Policy for how to exchange system load information among load.
  5. Policy to assign priority of execution of local and remote process on particular node.
  6. Migration limiting policy to determine number of times process can migrate from one node to other.
- If process is processed at its originating node then it is **local** process. If process is processed on node which is not its originating node then it is **remote** process. If process arrived from other node and admitted for processing then it becomes local process on this current node. Otherwise it is sent to other nodes across the network. It then is considered as remote process at destination node.

Load Estimation Policy

- Load-balancing algorithm should first estimate the workload of the particular node based on some measurable parameters. These parameters include factors which are time dependent and node-dependent. These factors are: Total number of processes present on node at the time of load estimation, resource demands of these processes, instruction mixes of these process and architecture and speed of the processors.
- As current state of the load is important, estimation of the load should be carried out in efficient manner. Total number of processes present on node is considered to be inappropriate measure for estimation of load. This is because actual load could vary depending on remaining service time for those processes. Again, how to measure remaining service time of all processes is the problem.
- Again both measures, total number of processes and total remaining service time of all processes are not suitable to measure load in modern distributed system. In modern distributed system, idle node may have several processes such as daemons, window managers. These daemons wakeup periodically and again sleeps.
- Therefore, CPU utilization is considered as best measure to estimate the load of the node. CPU utilization is number of CPU cycles actually executed per unit of real time. Timer is set up to observe CPU state time to time.

Process Transfer Policies

- In order to balance load in system, process is transferred from heavily loaded nodes to lightly loaded nodes. There should be some policy to decide whether node is heavily or lightly loaded. Most of the load-balancing algorithms use **threshold policy.**
- At each node, threshold value is used. Node accepts new process to run locally if its workload is below threshold value. Otherwise process is transferred to another lightly loaded node. Following policies are used to decide threshold value.
  - **Static policy :** Each node has predefined threshold value as per its processing capabilities. It remains fixed and does not vary with changing workload at local or remote node. No exchange of state information is needed in this policy.
  - **Dynamic policy :** In this policy, threshold value for each node is product of average workload of all the nodes and predefined constant for that node.
  - Predefined constant value c_k for node n_k is depends on processing capability of node n_k with respect to processing capability of all other nodes. Hence state information exchange between nodes is needed in this policy.

- Most of the load-balancing algorithms use **single threshold policy** which only defines two states of the node Overloaded and underloaded. In this policy, if load is below threshold value then node is considered as underloaded and it accepts new local or remote processes for execution. In case of load above the threshold value, nodes transfer the local processes and reject to accept remote processes.

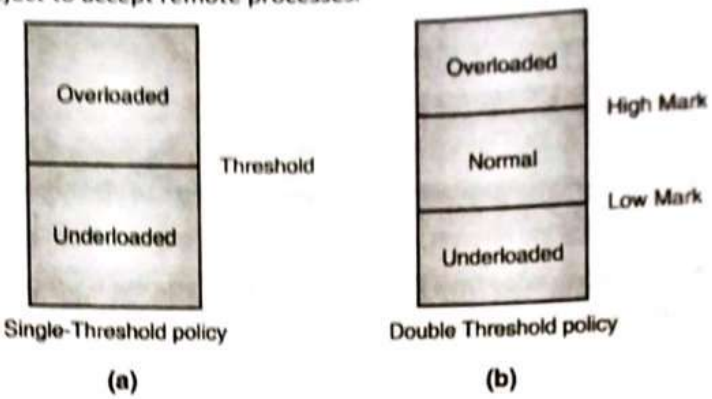


Fig. 4.4.1

- In single threshold policy, suppose load is just below the threshold value and node accepts remote process. In this case, load becomes larger than threshold value as soon as remote process is accepted. Immediately after accepting the remote process, node will start transferring its process too other nodes. This makes scheduling algorithm unstable.
- Therefore it is suggested that, node should transfer its processes to other nodes if this transfer greatly improves performance of its local processes. Node should also accept remote processes provided these newly accepted processes do not much affect the service to local processes.
- In order to minimize instability of single threshold policy, a **double-threshold policy** uses two threshold values high mark and low mark. It defines three states about load on a node: overloaded, normal and underloaded as shown in Fig. 4.4.1 (b).
- When load is in overloaded region, new local processes are sent to other nodes and request to accept remote processes are rejected. When load is in Normal region, new local processes run locally and request to accept remote processes are rejected. When load is in underloaded region, new local processes run locally and request to accept remote processes are accepted.

Location Policies

- Location policy of scheduling algorithm decides selection of destination node to transfer process. Following are different types of policies.
- **Threshold :** Any random node is selected and check is made to see whether transfer of process to this node would prohibit it to accept remote processes. If no, then process is transferred to this selected node. If yes, then another node is probed in similar way. This carries on till suitable node is located or number of probes exceeds static probe limit. In later case, process is executed on origin node. This policy gives good performance for small and large probe limit also.
- **Shortest :** In this method, random amount of nodes are chosen and each is polled to check its load. Process is transferred to the node having minimum load provided its load is such that which would not prohibit it to accept remote processes. If none of the polled node can accept load then process is executed on its originating node. If process is transferred to selected destination node then it (destination node) should execute the process despite of its state when process arrives. Probing is stopped if node with zero load is located. This is simple improvement in basic policy. In this policy, more state information exchange is needed.

- **Bidding :** In this policy, bidding process is carried out in which each node plays two roles; manager and contractor. Manager node has processes to send for execution and contractor is a node which is ready to accept processes for execution. Same node can play both the roles. Manager broadcasts request-for-bid message to all the nodes and contractor return bid with message containing memory size, processing capability, resource availability and other. Manager determines best bid upon receiving bid from contractor and sends process there. It may also happen that, contractor receives many bids from many managers and becomes overloaded. Solution to this problem is that, manager sends message to contractor before transferring the process. Contractor replies with message whether process will be accepted or rejected. In this policy, both managers and contractor have freedom to send and accept/reject processes respectively. Drawback is more communication overhead and difficult to decide best pricing policy.
- **Pairing :** In this policy, two nodes that greatly differs in their load are temporarily paired each other. Process transfer then is carried out from heavily loaded node to lightly loaded node. There may be many pairs established in the system and simultaneous transfer of processes is carried out among the pairs.

State Information Exchange Policies

- It is necessary to choose best policy to support dynamic load balancing algorithms and to reduce number of messages transmitted.
- Following policies are used to exchange state information.
- **Periodic Broadcast :** In this policy, each node broadcast its state information after elapse of every t units of time. It generates heavy network traffic and unnecessary messages get transferred although state is not changed. It has poor scalability.
- **Broadcast When State Changes :** In this method, node broadcast its state information only if its state changes. Node's state changes if it transfers its processes to other nodes or if it receives processes from other nodes. Further improvement to this method is to do not report small change in state to other nodes as nodes takes part in load balancing process if they are underloaded or overloaded. Refined method is node will broadcast its state information when it moves from normal load to underloaded or overloaded state.
- **On-Demand Exchange :** In this method, node broadcast its state information request when its state moves from normal load to either underloaded or overloaded region. Receivers of this request, replies their current state to requesting node. This method work with double-threshold policy. In order to reduce number of messages, further improvement is to avoid reply regarding current state from all the nodes. If requesting node is underloaded then only overloaded nodes can cooperate with it. If requesting node is overloaded then only underloaded nodes can cooperate with it in load-balancing process. As a result, in this improved policy requesting nodes identity is included in state information request message so that only nodes which can cooperate with requesting node sends reply. In this case, other nodes do not sends any reply to requesting nodes.
- **Exchange by polling :** This method do not use broadcast which limits the scaling. Instead, the node which needs cooperation from other nodes for balancing the load searches for suitable partner by polling other nodes one by one. Hence, exchange of state information takes place between polling node and polled node. Polling process stops either if appropriate node is located or predefined poll limit is attained.

Priority Assignment Policies

- Main issue in scheduling the local or remote processes at particular node in load balancing process is deciding the priority assignment rules. Following rules are used to assign priority to the processes at particular node.
  - **Selfish :** In this rule, local processes gets higher priority compared to remote processes.
  - **Altruistic (Unselfish) :** In this rule, remote processes are given higher priority compared to local processes.

- **Intermediate :** In this rule, priority is determined on the basis of number of local processes and number of remote processes at that particular node. This rule assigns higher priority to local processes if their total number is greater or equal to number of remote processes. If not, remote processes are assigned with higher priority.

Migration Limiting Policies

- Total number of times the process should migrate from one node to other is also important in balancing the node of system. Following two policies may be used.
- **Uncontrolled :** Process can be migrated any number of times from one node to other. This policy leads to instability.
- **Controlled :** As process migration is costlier operation, and hence, this policy limits migration count to 1. This means process should be migrated to only one node. For large size processes this count can be increased as process cannot finish execution in specified time on the same node.

4.5 Load-Sharing Approach

- In load-balancing approach, exchange of state information among nodes is required. This involves considerable overhead. Since resources are distributed, load cannot be equally distributed among all nodes. There should be a proper utilization of these resources. In large distributed system, number of processes in a node always fluctuating. Hence, temporal unbalancing of load among nodes always exists.
- Therefore, it is sufficient to keep all the nodes busy. It should ensure that, no node will remain idle while other nodes have several processes to execute. This is called as dynamic load sharing.

4.5.1 Issues in Designing Load-Sharing Algorithms

Load sharing algorithms aims at no node is idle while other nodes are overloaded. In load sharing algorithms, issues are similar to issues in load-balancing algorithms. It is simpler to decide about policies in load-sharing algorithms with compare to load-balancing algorithms.

Load Estimation Policies

- Load-sharing algorithms only ensure that no node is idle. Hence, simplest policy to estimate load is total number of processes on node.
- As in modern distributed system, several processes permanently resides on idle node, CPU utilization is considered as measure to estimate load on node

Process Transfer Policies

- Load-sharing algorithms mainly concern with two states of nodes: busy or idle. It uses single threshold policy with threshold value of 1. Node accepts process if it has no process. Node transfers the process as soon as it has more than two processes. If node becomes idle, it can be unable to accept new process immediately. This policy wastes the processing power of idle node.
- The solution to above problem is to transfer the process to the node which is expected to become idle. Therefore, some load-sharing algorithms use single threshold policy with threshold value of 2. If CPU utilization is considered as measure of load then double threshold policy should be used.

Location Policies

- In load-sharing algorithms, sender or receiver node is involved in load sharing process.

- **Sender Initiated Policy :** In this location policy, sender takes decision about transfer of process to particular node. In this policy, heavily loaded node search for lightly loaded nodes to transfer the process. When load of the node becomes greater than threshold value, it either broadcast or randomly probes the other nodes to find lightly loaded node which can accepts one or more of its processes.
- The good candidate node to accept the processes will be that node whose load would not exceed threshold value after accepting the processes for execution. In case of broadcast, sender node immediately knows about availability of suitable node to accept processes when it receives reply from all the receivers. Whereas, in random probing, probing continuous till appropriate receiver is found or number of probes reaches to static probe limit. If no suitable node is found then process is executed on its originating node. Probing method is more scalable than broadcast.
- **Receiver Initiated Policy :** In this location policy, receiver takes decision about from where to get the process. In this policy, lightly loaded node search for heavily loaded nodes to get load from there. When load of the node goes below the threshold value, it either broadcast or randomly probes the other nodes to find heavily loaded node which can transfer one or more its processes.
- The good candidate node to send its processes will be that node whose load would not reduce below threshold value after sending the processes for execution. In case of broadcast, sender node immediately knows about availability of suitable node to send processes to it when it receives reply from all the receivers. Whereas, in random probing, probing continuous till appropriate node is found from which processes can be obtained or number of probes reaches to static probe limit.
- In later case, node waits for fixed timeout period before trying again to initiate transfer. Probing method is more scalable than broadcast.
- Preemptive-process migration is costlier than non-preemptive as execution state needs to be transfer along with process. In non-preemptive process migration, process is transferred before it starts the execution on its current node. Receiver initiated process transfer is mostly preemptive. Sender initiated process transfer is either preemptive or non-preemptive.

State Information Exchange Policies

- In load-sharing approach state information exchange is carried out only when node changes the state. Because, node needs to know state of other node when it becomes overloaded or underloaded. In this case following policies are used.
- **Broadcast When State Changes :** In this policy, node broadcast state information request when its state changes. In sender initiated policy, this message broadcast is carried out when node becomes overloaded and in receiver initiated policy, this message broadcast is carried out when node becomes underloaded.
- **Poll When State Changes :** Polling is appropriate method compared to broadcast in large networks. In this policy, upon change in state, node polls other nodes one by one and exchanges its changed state information with them. Exchanging the state information continuous till appropriate node for sharing load is found or number of probes reaches to probe limit. In sender initiated policy, probing is carried by overloaded node and in receiver initiated policy, probing is carried out by underloaded node.

4.6 Introduction to Process Management

- Process management includes different policies and mechanisms to share processors among all the processes in system.

- Process management should ensure best utilization of processors by allocating them to different processes in system. Process migration deals with transfer of process from its current location to the node or processor to which it has been assigned.

4.7 Process Migration

- Process migration is transfer of process from its current node which is source node to destination node. There are two types of process migration. These are given below.
  - **Non-preemptive process migration :** In this approach, process is migrated from its current to destination node before it starts executing on current node.
  - **Preemptive process migration :** In this approach, process is migrated from its current to destination node during its course of execution on current node. It is costlier than non-preemptive process migration as process environment also needs to transfer along with process to destination node.
- Migration policy includes selection of process to be migrated and selection of destination node to which process should be migrated. This is already described in resource management. Process migration describes actual transfer of process to destination node and mechanism to transfer it.
- **Freezing time** is the time period for execution of process is stopped for transferring its information to destination node.

4.7.1 Desirable Features of Good Process Migration Mechanism

Following are desirable features of good process migration mechanism.

- **Transparency :** Following level of transparency is found.
  - **Object Access Level :** It is minimum requirement for system to support non-preemptive process migration. Access to objects like file and devices should be location independent. This allows to initiation of program on any node. To support transparency at object access level, system should support for transparent object naming and locating.
  - **System Calls and IPC :** System calls and interprocess communication should be location independent. Migrated process should not depend on its originating node. For preemptive process migration, system should support this transparency. Process should receive its messages from other processes directly on recently migrated node and not through its originating node.
- **Minimal Interference :** Migration of process should not affect the progress of process and system as a whole. This can be achieved by minimizing the freezing time of process being migrated.
- **Minimal Residual Dependency :** Migrated process should not continue to depend on its previous node once it has started its execution on its new node. Otherwise, failure of previous node will cause the process to fail. Also, it imposes load on its previous node.
- **Efficiency :** Process migration will be inefficient if it takes more time for migrating the process, more cost of locating the object and involves more cost to support remote execution once process is migrated.
- **Robustness :** Failure of other node (excluding current node on which process is running) should not affect accessibility and execution of that process.
- **Communication between Coprocesses of a Job :** If processes of the single job are distributed over several nodes for parallel execution then these processes should be able to directly interact with each other irrespective of their location.

4.7.2 Process Migration Mechanisms

Following activities are involved in process migration.

1. Freezing the process at source node and restarting its execution on its destination node.
2. Transferring the process's address space from its source node to its destination node.
3. Forwarding all the messages for process on its destination node.
4. Handling communication between cooperating processes placed on different nodes due to process migration.

Mechanism for Freezing and Restarting the Process

- In preemptive process migration, usually snapshot of the process state is taken at its source node and then this snapshot is reinstated at its destination node. First process execution is suspended at its source node. Its state information is then transferred at destination node. After this, its execution is restarted at its destination node.
- Freezing the processes at source node means its execution is suspended and all external interaction with the process is postponed.
- **Immediate and Delayed Blocking of the Process :** In preemptive process migration, execution of the process must be blocked at its source node. This blocking can be carried out immediately or after process reaches the state when it can be blocked. Hence, when to block the process depends on its current state. If process currently not executing system call then it can be immediately blocked. If process currently executing system call but is sleeping at interruptible priority waiting for occurring of kernel event then it can be immediately blocked. If process currently executing system call but is sleeping at non-interruptible priority waiting for occurring of kernel event then it cannot be blocked immediately. The mechanism of blocking can vary as per implementation.
- **Fast and Slow I/O Operations :** Allow completing all fast I/O operations such as disk I/O, before freezing the process at source node. It is not feasible to wait for slow I/O operations such as those on pipes and terminal.
- **Information about Open Files :** State information of process also contains information about files currently open by process. It includes name or identifier of the files, access modes, current position of their file pointers etc. In distributed system as network transparent execution environment is supported, this state information can be collected. Same protocol is used to access local or remote file. The UNIX based system recognizes the file by its full pathname. Operating system returns file descriptor to process once files is opened by it. It then uses file descriptor to complete I/O operations on file. In such system, pointer to file must be preserved so that migrated process can keep on accessing the file.
- **Reinstating Process on its Destination Node :** New empty process is created on destination node which is same as that allocated during process creation. Identifier of this newly allocated process can be same or different depending on implementation. If it is different, then it is changed to the original identifier in a subsequent step before process starts executing on destination node. Operations on both the processes are suspended. Hence, rest of the system cannot notice existence of two copies. When entire state of migrating process is transferred and copied in empty process, the new process is unfrozen and old is deleted. Thus process restarts execution from the state it has before being migrated. Migration mechanism should allow performing system call at destination node which was not completed due to freezing of the process at source node.

Address Space Transfer Mechanism

- Following information is transferred when process is migrated from source node to destination node.
  - **Process State :** It includes execution status i.e. content of registers, scheduling information, memory used by process, I/O states, and its rights to access the objects which is capability list, process's identifier, user and group identifier of the process, information about file opened by process.
  - **Process Address Space :** It includes code, data and stack of the program.
- Size of process state information is in few kilobytes. Whereas, process address space is of large size. Process address space can be transferred without stopping its execution on source node. However, while transferring state information, it is necessary to stop execution of the process.
- To start execution at destination node, state information is needed at destination node. Address space can be transferred before or after process starts execution at destination node.
- Address space can be transferred in following manners.

Total Freezing

- In this method, process is not allowed to continue its execution when decision of its transfer to destination is taken. The execution of the process is stopped while its address space is being transferred. This method is simple and easy to implement.
- The limitations of this method is that if process that is to be transferred is suspended for longer period of time during process of migration then timeouts may occur. Also user may observe delay if the process is interactive.

Pretransferring

- In this method, process is allowed to run on its source node until its address space has been transferred on destination node. The modified pages at source node during transfer of address space also are retransferred to the destination followed by previous transfer. This retransfer of modified pages is carried out repeatedly until number of modified pages is relatively small. These remaining modified pages are transferred after the process is frozen for transferring its state information.
- In first pretransfer operation, as first transfer of address space takes longest time, a longest time is offered accordingly to carry out modifications related to changed pages. Obviously, second transfer will take short time for the same as only modified pages during first transfer is transferred. These modifications will be lesser compared to first transfer. Subsequently, fewer pages will be transferred in next transfer to destination node till number of pages converges to zero or very few.
- These remaining pages are then transferred from source node to destination node when process is frozen. At source node, pretransfer operation gets higher priority with respect to other programs. In this method, frozen time is reduced and hence, leads to minimal interfere of migration with interactions of process to other processes and users. As same pages can be transferred repetitively due to modifications, the total time elapsed in migration can increase.

Transfer on Reference

- In this method, entire address space of process is kept on its source node and only needed address space for execution is requested from destination node during execution (just like relocated process executes at destination node) . The page is transferred from source node on its reference at destination node. This is just like demand driven copy on reference approach.
- This method is more dependent on source node and imposes burden on it. Moreover, if source node fails or reboots the process execution will fail.

Message-Forwarding Mechanism

- It should be ensured that, the migrated process should receive all its pending, messages in transmission (en-route messages) and future messages at destination node. Following types of messages needs to be forwarded at destination node of migrated process.

Type 1 : Messages which are received at source node after process's execution has been stopped and its execution has not until now been commenced at destination node.

Type 2 : Messages received at source node after process's execution started at destination node.

Type 3 : Messages expected by already migrated process after it starts its execution on destination node.

Mechanism of Resending the Message

- This mechanism is used in V-System and amoeba distributed OS to handle above all types of messages. In type 1 and 2 case, messages are simply returned to sender as they are not deliverable or simply dropped with assumption that, sender maintained the copy of them and able to retransmit it. Messages of type 3 are directly sent to the destination node of the process.
- In this method, there is no need to maintain the process state at source node. But, message forwarding mechanism of process migration operation is not transparent to the other processes communicating with the migrant process.

Original Site Mechanism

- In this method, the process identifier has its home node embedded in it and every node is responsible to maintain information about current location of all the processes created on it. Hence, current location of the process can be obtained from its home node. Any process now first forwards the message to home node which then forwards message to process's current node.
- The drawback of this method is that, it imposes continuous load on process's home node although process is migrated to other node. Also, failure of home node disturbs the message forwarding mechanism.

Link Traversal Mechanism

- In this method, message queue of the migrant process is created on its source node. All the type 1 messages are placed in this queue. After knowing that the process is established at destination node, all the messages in message queue are forwarded to the destination node.
- For Type 2 and 3 messages, a link pointing to destination node of the migrant process is maintained at source node. This link is address for forwarding processes. This link consists of two components: a system wide unique process identifier which is identifier of the originating node of the process and other is unique local identifier which is a last known location of a process. First component remains same during lifetime of the process. Whereas second component may change. A series of links, starting from node where process is created, is traversed in order to forward Type 2 and 3 messages. This traversal stops at process's current node and second component of the link is now updated when process is accessed from the node. This improves efficiency next time when process is located from the same node.
- The drawback of this method is that, failure of any node in the chain fails then process cannot be located. It has poor efficiency as many links needs to be traversed to locate the process.

Mechanism for Handling Coprocesses

There should be efficient communication between parent and child processes if they are migrated and placed at different nodes.

Not Allowing Separation of Processes

- Processes that wait for one or more of their children to complete are not allowed to migrate. It is used by UNIX based networked system. The drawback of this approach is that, it does not permit use of parallelism within job. It is due restricting the various tasks of job to distribute on different nodes.
- If parent process migrates then its children processes will be migrated along with it. It is used by V-System. The drawback of this approach is that, it involves more overhead in migrating the process if logical host comprising the address space of V-System and their associated processes is large. It means logical host comprises several host processes.

Home Node in Original Sight Concept

- In this method, process and its children are allowed to migrate independently on different nodes. All the communications between parent process and its children takes place via home node.
- Drawback is that message traffic and communication cost increases.

4.7.3 Process Migration in Heterogeneous System

- In heterogeneous environment, it is necessary to translate data format when process is migrated to destination node. CPU format at source node and at destination node can be different. If n numbers of CPUs are present in the system then n(n–1) pieces of translation software is required.
- Solution to reduce this software complexity is to use external data representation in which each processor has to convert data to and from the standard form. Hence, if 4 processors are present then only 8 conversions are required. Two conversions for each processor, where one is to convert data from CPU format to external format and other in reverse order (external format to CPU format).
- In this method, translation of floating point numbers which consist of exponent, mantissa and sign needs to be handled carefully. In handling of exponent example suppose processor A uses 8 bit, B uses 16 bits and external data representation is designed for processor A offers 12 bits. Hence, process can be migrated from A to B which involves conversion of 8 bits to 12 bits and a12 bits to 16 bits for processor B. Process requiring exponent data more than 12 bits cannot be migrated from processor B to A. Therefore, external data representation should be designed with number of bits at least as longest exponent of any processor in the system.
- Migration from processor B to processor A also not possible if exponent is less than or equal to 12 bits but greater than 8 bits. In this case, although external data representation has sufficient number of bits but processor A does not.
- Suppose processor A uses 32 bits, B uses 64 bits and external data representation uses 48 bits. In this case, handling mantissa is same as handling exponent. Only transferring process from B to A will result in computation with half precision. This may not be acceptable when accuracy of result is important. Second problem may occur due to loss of precision due to multiple migrations between set of processors. Solution is to design external data representation with longest mantissa of any processor. External data representation also should take care of signed-infinity and signed-zero.

4.7.4 Advantages of Process Migration

Following are the advantages of process migration.

- **Reducing average response time of processes :** The response time of processes increases as load on the node increases. In process migration, as processes are moved from heavily loaded node to idle or lightly loaded node, the average response time of processes reduces.

- **Speeding up individual job :** The tasks of the job can be migrated to different node. Also job can be migrated to node having faster CPU. In both cases, execution of the job speeds up.
- **Gaining higher throughput :** As all the CPUs are better utilized with proper load balancing policy, higher throughput can be achieved. By properly mixing CPU and I/O bound jobs globally, throughput can be increased.
- **Effective utilization of resources :** In distributed system, hardware and software resources are distributed on many nodes. These resources can be effectively used by migrating processes to the nodes as per their resource requirements. Moreover, if processes require special-purpose hardware then they can be migrated to node having such hardware facility.
- **Reducing network traffic :** Migrating processes closer to the resources or to the node having their required resources, reduces network traffic as resources are accessed locally. Instead of accessing huge database remotely, it is better to move the process to the node where such database is present.
- **Improving system reliability :** Critical process can be migrated to node with higher reliability in order to improve reliability.
- **Improving system security :** This can be achieved by migrating sensitive process to more secure node.

4.8 Threads

- Threads improve performance of the application. In operating system supporting thread facility, basic unit of CPU utilization is threads. A thread is a single sequence stream within a process. Threads are also called as lightweight processes as it possess some of the properties of processes. Each thread belongs to exactly one process.
- In operating system that support multithreading, process can consist of many threads. These threads run in parallel improving the application performance. Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads can share common data so they do not need to use interprocess communication. Like the processes, threads also have states like ready, executing, blocked etc. priority can be assigned to the threads just like process and highest priority thread is scheduled first.
- Each thread has its own Thread Control Block (TCB). Like process context switch occurs for the thread and register contents are saved in (TCB). As threads share the same address space and resources, synchronization is also required for the various activities of the thread.

4.8.1 Comparison between Process and Thread

- Threads are more advantageous with compare to process. Following table shows comparison between thread and process.

Sr. No.	Process	Thread
1.	Program in execution called as process. It is heavy weight process.	Thread is part of process. It is also called as light weight process
2.	Process context switch takes more time as compared to thread context switch because it needs interface of operating system.	Thread context switch takes less time as compared to process context switch because it needs only interrupt to kernel only.
3.	New Process creation takes more time as compared to new thread creation.	New thread creation takes less time as compared to new process creation.

Sr. No.	Process	Thread
4.	New Process termination takes more time as compared to new thread termination.	New thread termination takes less time as compared to new process termination.
5.	Each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
6.	In process based implementation, if one process is blocked no other server process can execute until the first process unblocked.	In multithreaded server implementation, if one thread is blocked and waiting, second thread in the same process could execute.
7.	Multiple redundant processes use more resources than multiple threaded process.	Multiple threaded processes use fewer resources than multiple redundant processes.
8.	Context switch flushes the MMU (TLB) registers as address space of process changes.	No need to flush TLB as address space remains same after context switch because threads belong to the same process.

4.8.2 Server Process

Following models can be used to construct server process. Consider the example of file server.

- **As a Single Threaded process :** In this model client requests are served sequentially. There is no parallelism. The model uses blocking system call. File server gets client's file access request from request queue. If access is permitted then server checks if disk access is needed. If no, request is served immediately and reply is sent to client process. If disk access is needed then file server sends disk access request to disk server and waits till reply arrives. After getting disk server's reply, it services client request and sends reply to client process. Server then goes back to handle next request. Hence, performance server is unacceptable.
- **As a Finite-State Machine :** It supports parallelism with non-blocking call. Event queue is maintained to store disk server's reply messages and client's requests. If server remains idle, it takes next request from event queue. If it is client's request and if access is permitted then server checks if disk access is needed. If disk access is needed then file server sends disk access request to disk server. At this point, instead of blocking, it saves current state of client's request in table and goes to next request in event queue. This message may be new client's request or reply of previous request from disk server. If it is new client's request then it is handled in above manner and if it is reply from disk server then it retrieves current state of client's request in table. It the further processes this client request with retrieved client's current state.
- **As a Group of Threads :** This model supports parallelism with blocking system calls. In this model, server comprises one dispatcher thread and several worker threads. Worker threads can be created dynamically to handle the requests or pool of threads can be created at start up time. The dispatcher thread keeps waiting for the request. if request arrives and access is permitted then it creates new thread or chooses idle thread from available pool to handover this request. Now dispatcher changes state from running to ready. Worker thread checks if disk access is needed. It first checks if request is satisfied from block cache shared by threads. If not, then it sends disk access request to disk server and blocks waiting for reply from disk server. Now scheduler may chose another thread either dispatcher or worker which are ready to run.

4.8.3 Models for Organizing Threads

Following are the three different ways to organize the threads :

- **Dispatcher-worker Model :** In this model, process consist of one dispatcher thread and several worker threads. The dispatcher thread accepts the client's request chooses one of the free worker thread to handover this request for further processing. Hence, multiple client's request can be processed by multiple worker threads in parallel.
- **Team Model :** In this model, all thread are equal level. Each thread accepts client's request and process it on its own. A specific type of thread to handle specific requests can be created. Hence, parallel execution of different client's requests can be carried out.
- **Pipeline Model :** This model is useful for the applications which are based on producer consumer model. In this model, output of the first thread is given as input to second thread for processing, the output of the second thread is given as input to third thread for processing and so on. Threads are arranged in pipeline. In this way, output of the last thread is final output of the process to which threads belongs.

4.8.4 Issues in Designing a Thread Package

Following are the issues related designing a thread package. System should supports primitives for thread related operations.

Thread Creation

- Threads can be created statically in which number of threads remains fixed till lifetime of the process. In dynamic creation of threads, initially process is started with single thread. New threads are created when needed during the execution of process. A thread may destroy itself after finishing of its job by using exit system call.
- In static approach, number of threads is decided while writing or compiling the program. In this approach, a fixed stack is allocated to the thread. In dynamic approach, stack size is specified through parameter to the system call.

Thread Termination

- A thread may destroy itself after finishing of its job by using exit system call. A kill command is used to kill thread from outside by specifying thread identifier as parameter. In many cases, threads are never killed until process terminates.

Thread Synchronization

- The portion of the code where thread may be accessing some shared variable is referred as the Critical region (CR). It is necessary to prevent multiple threads accessing same data. For this purpose ***mutex*** variable is used. Thread that wants to execute in CR performs lock operation on corresponding mutex variable. In single atomic operation, state of mutex changes from unlocked to locked state.
- If mutex variable is already in locked state then thread is blocked and waits in queue of waiting threads on the mutex variable. Otherwise thread carries out other job but continuously retries to lock the mutex variable. In multiprocessor system where threads run in parallel, two threads may carry out lock operation on same mutex variable. In this case, one thread waits and other wins. To exit the CR, thread carries out unlock operation on mutex variable.
- Condition variables are used for more general synchronization. Operations wait() and signal() are provided for conditional variable. When thread carries out wait operation on condition variable the associated mutex variable is unlocked and thread is blocked till signal operation is carried out by other thread on the condition variable. When thread carries out signal operation on condition variable the mutex variable is locked and blocked thread on condition variable starts execution.

4.8.5 Thread Scheduling

Thread packages supports following scheduling features.

- **Priority Assignment Policy :** Threads are scheduled with simple FIFO or in round-robin (RR) basis. Application programmers can assign priority to threads so that important threads should get highest priority. This priority based scheduling can be non-preemptive or preemptive.
- **Flexibility to vary quantum size dynamically :** In multiprocessor system, if there are fewer runnable threads than available processors then it is wasteful to interrupt the running thread. Hence, instead of using fixed length quantum, variable quantum is used. In this case, size of time quantum is inversely with number of threads in the system. It gives good response time.
- **Handoff Scheduling :** Currently running thread may name its successor to run. It may give up processor and request the successor to run next, bypassing the queue of runnable threads. It enhances performance if cleverly used.
- **Affinity Scheduling :** In this scheduling in multiprocessor system, a thread is scheduled on processor on which it was last run, expecting that its address space is still in that processor cache.

4.8.6 Implementing a Thread Package

Thread Package can be implemented either in user space or kernel.

Implementing Threads in User Space

User Level Threads

- In user level implementation, kernel is unaware of the thread. In this case, thread package entirely put in user space. Java language supports threading package.
- User can implement the multithreaded application in java language. Kernel treats this application as a single threaded application. In a user level implementation, all of the work of thread management is done by the thread package.
- Thread management includes creation and termination of thread, messages and data passing between the threads, scheduling thread for execution, thread synchronization and after context switch saving and restoring thread context etc.
- Creation and destroying of thread requires less time and is chip operation. It is the cost of allocating memory to set up a thread stack and deallocating the memory while destroying the thread.
- Both operations requires less time. Since threads belong to the same process, no need to flush TLB as address space remains same after context switch. User-level threads requires extremely low overhead, and can achieve high computational performance. Fig. 4.8.1 shows the user level threads.

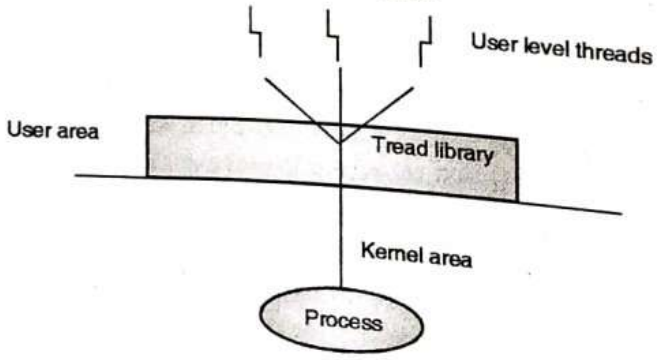


Fig. 4.8.1 : User level threads

Advantages of user level threads

- Thread switching does not require flushing of TLB and doing CPU accounting. Only value of CPU register need to be stored and reloaded again.
- User level threads are platform independent. They can execute on any operating system.
- Scheduling can be as per need of application.
- Thread management are at user level and done by thread library. so kernels burden is taken by threading package. Kernels time is saved for other activities

Disadvantages of user level threads

- If one thread is blocked on I/O, entire process gets blocked.
- The applications where after blocking of one thread other requires to run in parallel, user level threads are of no use.

Implementing Threads in Kernel

Kernel Level Threads

- In this, threads are implemented in operating system's kernel. The thread management is carried out by kernel.
- All these thread management activities are carried out in kernel space. So thread context and process context switching becomes same.
- Application can be written as multithreaded and threads of the application are supported as threads in single process.
- Kernel threads are generally requires more time to create and manage than the user threads.

Advantages of kernel level threads

- The kernel can schedule another thread of the same process even though one thread in a process is blocked. Blocking of one thread does not block the entire process.
- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.

Disadvantages of kernel level threads

- Context switch requires kernel intervention.
- Kernel threads are generally requires more time to create and manage than the user threads.

Hybrid Implementation

- Considering the advantages of user level and kernel level threads, a hybrid threading model using both types of threads can be implemented.
- The Solaris operating system supports this hybrid model. In this implementation, all the thread management functions are carried out by user level thread package at user space. So operations on thread do not require kernel intervention.
- The advantage of hybrid model of the threads is that, if the applications are multithreaded then it can take advantage of multiple CPUs if they are available.
- Other threads can continue to make progress even if the kernel blocks one thread in a system function.

4.9 Virtualization

- From availability and security point of view, using separate computer to put each service is more preferable by organizations. If one server fails, other will not be affected.

- It is also beneficial in case if organizations need to use different types of operating system. However, keeping separate machines for each service is costly. Virtual machine technology can solve this problem. Although this technology seems to be modern but idea behind it is old.
- The VMM (Virtual Machine Monitor) creates the illusion of multiple (virtual) machines on the same physical hardware. A VMM is also called as a hypervisor. Using virtual machine, it is possible to run legacy applications on operating systems no longer supported or which do not work on current hardware.
- Virtual machines permits to run at the same time applications that use different operating systems. Several operating systems can run on single machine without installing them in separate partitions. Virtualization technology plays major role in cloud computing.

4.9.1 The Role of Virtualization in Distributed Systems

- Practically, every machine offers a programming interface to higher level software, as shown in Fig. 4.9.1. There are many different types of interfaces, for example, instruction set as offered by a CPU, and as other example, huge collection of application programming interfaces that are shipped with many present middleware systems.

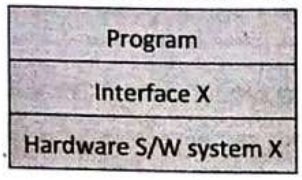


Fig. 4.9.1 : Program, Interface and System

- Virtualization makes it possible to extend or replace an existing interface so as to imitate the activities of another system, as shown in Fig. 4.9.2.

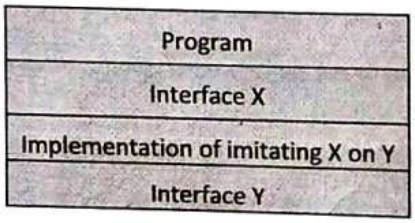


Fig. 4.9.2 : Virtualizing of system X on top of Y

- With advancement in technology, hardware and low-level system software change reasonably fast compared to applications and middleware which are at higher level. Hence, it is not possible to maintain pace of legacy software with the platforms it relies on. With virtualization, legacy interfaces can be ported to new platforms and hence, the latter can be opened for large classes of existing programs.
- In large distributed system, environment is heterogeneous. It comprises of heterogeneous collection of servers running different applications which are accessed by clients. Application should also use various resources easily and efficiently. In this case, virtualization plays major role.
- Each application can run on its own virtual machine, perhaps with the related libraries *and* operating system, which, in turn, run on a common platform. In this way, the diversity of platforms and machines can be minimized and high degree of portability and flexibility can be achieved.

4.9.2 Architectures of Virtual Machines

- Four types of interfaces are offered by any computer system. These are : An interface between the hardware and software, consisting of machine instructions that can be invoked by any program, interface between the hardware and software consists of machine instructions that can be invoked by OS only, an interface consisting of system calls and application programming interface (API).

- In first technique of virtualization, we can build a runtime system that basically offers an abstract instruction set that is to be used for executing applications. Instructions can be interpreted, but could also be emulated as is done for running Windows applications on UNIX platforms.
- In other technique of virtualization, a system that is basically implemented as a layer completely shielded the original hardware is provided and offered the complete instruction set of that same or other hardware as an interface. This interface can be offered *at the same time* to different programs. Hence, it is now possible to have multiple, and different operating systems run independently and in parallel on the same platform.

4.10 Clients

4.10.1 Networked User Interfaces

Client contact the server to access remote service. For this purpose, client machine may have separate counterpart that can contact the service over network. Alternatively, a convenient user interface can be offered to access service. Client need not have local storage and has only terminal. In the case of networked user interfaces, entire processing and storage is carried out at server.

Example : The X Window System

- It is used to control bit-mapped terminals, which include a monitor, keyboard, and a pointing device such as a mouse. It is just like component of an operating system that controls the terminal. The X kernel comprises all the terminal-specific device drivers. It is usually highly hardware dependent. The X kernel offers a relatively low-level interface for controlling the screen, but also for capturing events from the keyboard and mouse.
- This interface is made available to applications as a library called Xlib. The X kernel and the X applications may reside on the same or different machine. Particularly, X offers the X protocol, which is an application-level communication protocol by which an instance of Xlib can exchange data and events with the X kernel.
- As an example, Xlib can request to X kernel to create or kill a window, for setting colors, and defining the type of cursor to display, etc. The X kernel will react to local events such as keyboard and mouse input by sending event packets back to Xlib. Several applications can communicate simultaneously with the X kernel. Window manager application has special rights. This application can dictate the "look and feel" of the display as it appears to the user.

4.10.2 Client-Side Software for Distribution Transparency

- In client server computing, some part of processing and data level is executed at client-side as well. A special class is formed by embedded client software, such as for automatic teller machines, cash registers, barcode readers, TV set-top boxes, etc. In these cases, the user interface is a relatively small part of the client software, in contrast to the local processing and communication facilities.
- Client software also includes components for achieving distribution transparency. A client should not be aware that it is communicating with remote processes. On the contrary, distribution is often less transparent to servers for reasons of performance and correctness.
- As, at client side offers the same interfaces which are available at server, access transparency is achieved by client stub. The stub offers the same interface as available at the server. It hides the possible differences in machine architectures, as well as the actual communication.
- The location, migration, and relocation transparency is handled with different manner. Client request is always forwarded to all copy of server replica. Client-side software can transparently collect all responses and pass a single response to the client application. Failure transparency is also handled by client. Middleware carries out masking of communication failure. Client can repetitively attempt to connect the same server or other server.

4.11 Servers

4.11.1 General Design Issues

- Server takes request sent by client, processes it and sends response to client. Iterative server itself handles request and sends response back to client. Concurrent server passes incoming request to thread and immediately waits for another request. Multithreaded server is example of concurrent server.
- Server listen client request at endpoint. Client sends request to this end point called as port. These end points are globally assigned for well-known services. For example, servers that handle Internet FTP requests always listen to TCP port 21. Some services uses dynamically assigned endpoint. A time-of-day server may use an end point that is dynamically assigned to it by local OS.
- In this case, a client has to search the end point. For this purpose, a special daemon process is maintained on each server machine. The daemon keeps track of the current endpoint of each service implemented by a co-located server. The daemon itself listens to a well-known end point. A client will first contact the daemon, request the end point, and then contact the specific server.
- Another issue is how to interrupt the server while operation is going on. In this case, user suddenly exits the client application, immediately restart it, and pretend nothing happened. The server will finally tear down the old connection, thinking the client has most likely crashed. Other technique to handle communication interrupts is to design the client and server such that it is possible to send out-of-band data, which is data that is to be processed by the server before any other data from that client.
- One solution is to let the server listen to a separate control end point to which the client sends out-of-band data, while at the same time listening (with a lower priority) to the end point through which the normal data passes. Another solution is to send out-of-band data across the same connection through which the client is sending the original request.
- Other important issue is whether server should be stateless or stateful. A stateless server does not keep information on the state of its clients, and can change its own state without having to inform any client. Consider the example of file server. Client access the remote files from server. If server keeps track about each file being accessed by each client then the service is called stateful. If server is simply provides requested blocks of data to client and does not keep track about how client makes use of them then service is called stateless.

Stateful File Service

- First client must carry out open( ) operation on file prior to accessing it. Server the access information about file from disk and store in main memory.
- Server then gives unique connection identifier to client and opens the file for client. This identifier is used by client to access the file throughout the session.
- On closing of the file, or by garbage collection mechanism, the server should free the main memory space used by client when it is no longer active. The information maintained by server regarding client is used in fault tolerance. AFS uses stateful approach.

Stateless File Service

- In this case server does not keep any information in main memory about opened files. Here, each request identifies the file. There is no need to open and close the file with operations open( ) and close( ).
- Each file operation in this case is not a part of session but it is on its own separately. Closing the file at last is also a remote operation. NFS is example of stateless file service approach.

- Stateful service gives more performance. As connection identifier is used to access information maintained in main memory, disk accesses are reduced. Moreover, as server knows that file is open for sequential access, then read ahead next block improves the performance. In stateful case, if server crashes then recovery of volatile state information is complex. A dialog with client is used by recovery protocol. Server also needs to know about crash of client to free the memory space allocated to it. All the operation underway during crash should be aborted.
- In case of stateless service, the effect of server crash and recovery is invisible. Client keeps retransmitting the requests if server crash and no response from it.

4.11.2 Server Clusters

- Server cluster is collection of server machines that are connected through network. Consider these machines are connected through LANs which often offers high bandwidth. This server cluster logically organized in three tiers as shown in Fig. 4.11.1.
  - **Switch :** Client requests are routed through it.
  - **Application/Compute Servers :** These are high performance servers or application servers.
  - **Data Processing Servers :** These are file or database servers.
- The server cluster offering multiple services can have different machines running different application servers. Therefore, the switch should distinguish services. Otherwise it will not be possible to forward requests to the proper machines.

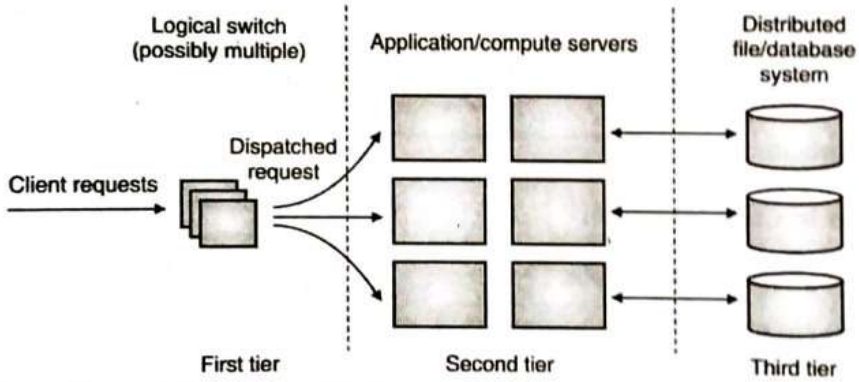


Fig. 4.11.1 : Logical organization of server cluster

- In server cluster, some of the servers may remain idle as it may be running single service. Hence, it is advantageous to temporarily migrate the services to idle machines. Client applications should always remain unaware about the internal organization of cluster.
- In order to access cluster, a TCP connection is set up over which application-level requests are sent as part of a session. A session ends by terminating the connection. If transport-layer switches are used, then switch accepts incoming TCP connection requests, and hands off such connections to one of the servers. In coming request comes to single address of switch which then forwards it to appropriate server.
- Server's ACK message to client contains IP address of switch as a source address. Switch follows the round robin policy to distribute load among servers. More advanced server selection criteria can be deployed as well.

4.11.3 Distributed Servers

- Using single switch (access point) to forward clients request to multiple servers may cause single point of failure. The cluster may become unavailable. As a result of this, several access points can be used, of which the addresses are made publicly available.
- A distributed server is dynamically varying collection of machines, with also possibly varying access points which appears to the outside world as a single powerful machine. With such a distributed server, the clients benefit from a robust, high-performing, stable server. The available networking services, particularly mobility support for IP version 6 (MIPv6) is used. In MIPv6, a mobile node normally resides in home network with its home address (HoA) which is stable.
- When mobile node changes network, it receives care-of-address in foreign network. This care-of address is reported to the node's home agent (router attached to home network) which forwards traffic to the mobile node. All applications communicate with mobile node using home address, they never see care of address.
- This concept can be used for stable address of distributed server. In this case, a single unique contact address is initially assigned to the server cluster. The contact address will be the server's life-time address to be used in all communication with the outside world. At any time, one node in the distributed server will work as an access point using that contact address, but this role can easily be taken over by another node.
- What happens is that the access point records its own address as the care-of address at the home agent associated with the distributed server. At that point, all traffic will be directed to the access point, which will then take care in distributing requests among the currently participating nodes. If the access point fails, a simple fail-over mechanism comes into place by which another access point reports a new care-of address. Home agent and access point may become bottleneck as whole traffic would flow these two machines. This situation can be avoided by using MIPv6 feature known as route optimization.

4.12 Code Migration

4.12.1 Approaches to Code Migration

Let us discuss about why to migrate the code.

Reasons for Migrating Code

- Performance is improved by moving code from heavily loaded machines to lightly loaded machine, although it is costlier operation. If server manages huge database and client application perform operations on this database that involves large quantity of data. In this case, it is better to transfer part of client application to server machine. In other case, instead of sending requests for data validation to server it is better to carry out client side validation. Hence, to transfer part of server application to client machine.
- Code migration also helps to improve performance by exploiting parallelism where there is no need to consider details of parallel programming. Consider the example of searching for information in the Web. A search query which is small mobile program (mobile agent) moves from site to site. Each copy of this mobile agent can be transferred to different sites to achieve a linear speedup compared to using just a single program instance. Code migration also helps to configure distributed system dynamically.
- In other case, no need to keep all software at client. Whenever client binds with server, it can dynamically download the required software from web server. Once work is done, all these software can be discarded. No need to preinstall

Models for Code Migration

- In first model, program is comprises of code segment, resource segment and execution segment. Code segment contains all the instructions of program. Resource segment includes references to external resources needed by the process, such as files, printers, devices, other processes, and so on. Execution state includes current execution state that contains private data, the stack, and program counter.
- In weak mobility, only code segment with initialization data is transferred. For example, Java applets always start execution from initial state. On the other hand, in strong mobility, execution segment is needed to resume execution on target node as execution of processes is stopped at source node and resumes at target node.
- Migration of code can be sender-initiated or receiver-initiated. In sender initiated migration, migration initiates from the node (source node) where the code is currently running. In receiver-initiated migration, target machine takes initiative for code migration. Receiver-initiated migration is simpler than sender-initiated migration. Some time client initiate migration and in the same way server initiates migration.
- In weak mobility, migrated code either is executed by the target process or a separate process needs to be started. For example, downloaded Java applets execute in the browser's address space. It avoids staring a separate process and hence, avoids communication at the target machine. The main drawback is that the target process needs to be protected against malicious or unintentional code executions.
- A simple solution is to let the operating system take care of that by creating a separate process to execute the migrated code. Migration by cloning is a simple way to improve distribution transparency in which process creates child processes.
- Following are the alternatives for code migration.

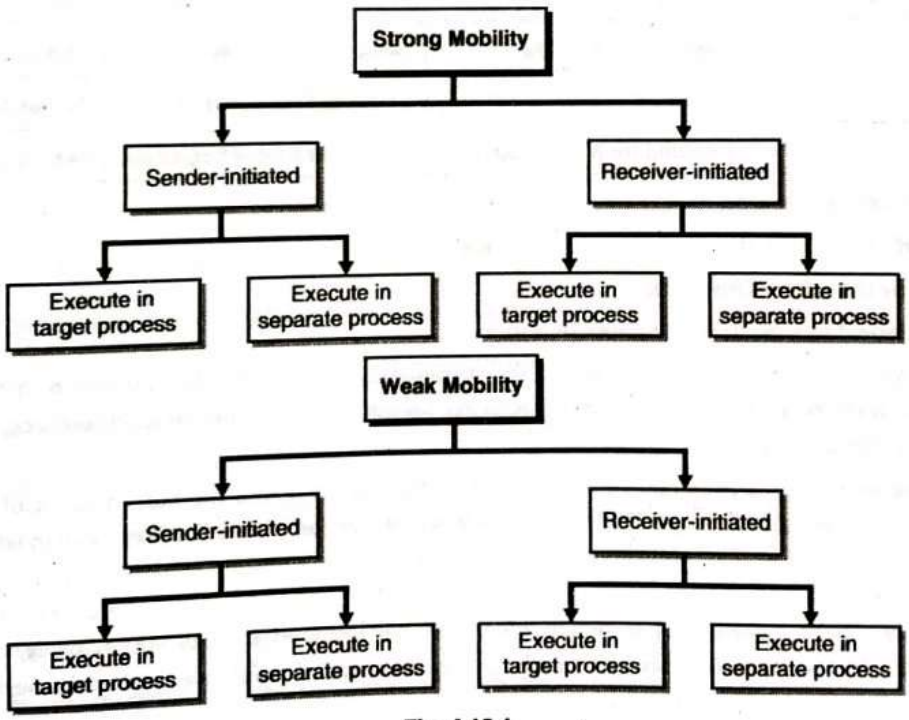


Fig. 4.12.1

4.12.2 Migration and Local Resources

- In migration of resource segment, suppose process holds a reference to a specific TCP port to communicate with remote processes. This reference is held in its resource segment. After migration to other node, process will have to give up the port and request a new one at the destination. In other cases such as file, reference which is URL will remain valid.
- Three types of **process-to-resource** bindings exist. **Binding by identifier** requires precisely the referenced resource such as URL to refer to a specific Web site or to FTP server by means of that server's Internet address. References to local communication end points are also in this category.
- In **binding by value,** only the value of a resource is needed. In this binding, execution of the process does not affect if same value is available on target node. A typical example of binding by value is libraries that are locally available, although their exact location in the local file system may differ between nodes. In **binding by type**, a process indicates it needs only a resource of a specific type. Example is: references to local devices, such as monitors, printers, and so on.
- It is also important to consider the **resource-to-machine** bindings. **Unattached resources** can be easily moved from one machine to other. The example is data file. On the other hand, it is possible to move **fastened resources** but it incurs high cost. Local databases and complete Web sites are the example of fastened resources. **Fixed resources** like local devices and communication endpoints cannot be moved.
- Considering above types of process-to-resource bindings and resource-to-machine bindings, following combination is required to consider while migrating the code.

Process to resource binding	Unattached Resources	Fastened Resources	Fixed Resources
By Identifier	Move (or Global Ref)	Global Ref (or Move)	Global Ref
By Value	Copy (or Move, Global Ref)	Global Ref (or Copy)	Global Ref
By Type	Rebind (or Move, Copy)	Rebind (or Global Ref, Copy)	Rebind (or Global Ref

  - **Move :** Move the resource.
  - **Global Ref :** Establish a global systemwide reference.
  - **Copy :** Copy the value of resource.
  - **Rebind :** Rebind process to locally available resource.
- It is best to move unattached resource along with the migrating code. If the resource is shared by other processes then establish a global reference such as URL to access resource remotely. For fastened or fixed resource, best solution is to create global reference.
- Establishing global reference is sometime expensive. For example, to access the huge amount of multimedia data. Suppose migrating a process is using a local communication end point which is a fixed resource to which the process is bound by the identifier. In this case, process set up a connection to the source machine after it has migrated and install a separate process at the source machine to forward all incoming messages. The drawback is that failure of source machine results in failure of communication with the migrated process. Alternatively, all processes that communicated with the migrating process should change their global reference, and send messages to the new communication end point at the target machine.
- In binding by value, fixed resource such as memory can be shared by establishing global reference. In this case, distributed shared memory support is needed.

- Runtime libraries are the example of binding by value with fastened resources. It can be available on the target machine, or should otherwise be copied before code migration takes place. In this case global reference is preferred if huge amount of data needs to be transferred.
- Unattached resources can be copied or moved to the new destination, if not shared by a many processes. If it is shared then, establishing a global reference is the only option. In case of bindings by type, the obvious solution is to rebind the process to a locally available resource of the same type.

4.12.3 Migration in Heterogeneous Systems

- In large distributed system, machine's architecture and platforms used are different. Hence, environment is heterogeneous. With the use of scripting languages and highly portable languages such as Java, solution to code migration in heterogeneous systems relies on virtual machine. In case of scripting languages, these virtual machines directly interpret source code. In case of Java, it interprets intermediate code generated by a compiler.
- Recently, instead of migrating only processes, migration of entire computing environment is preferred. It is possible to decouple a part from the underlying system and actually migrate it to another machine, provided proper compartmentalization is carried out. In this way, such migration supports strong mobility. In this type of migration, many complexities that arise in binding with different types of resources may be solved.

Review Questions

Q. 1 Explain in short scheduling techniques used in distributed system.

Q. 2 Explain desirable features of good global scheduling algorithm.

Q. 3 Explain in detail task assignment approach.

Q. 4 Explain and classify different load balancing algorithms.

Q. 5 Explain different issues in designing load-balancing algorithms.

Q. 6 Explain policy to estimate the load of a node in load balancing approach.

Q. 7 Explain different process transfer policies in load balancing approach.

Q. 8 Explain different approaches to locate the node in load balancing approach.

Q. 9 Explain different state information exchange policies in load balancing approach.

Q. 10 What are the different priority assignment techniques in load balancing approach? Explain.

Q. 11 Explain issues in designing load-sharing algorithms.

Q. 12 Explain different approaches to locate the node in load sharing approach.

Q. 13 Explain different state information exchange policies in load sharing approach.

Q. 14 Explain the types of process migration.

Q. 15 Explain desirable features of good process migration mechanism.

Q. 16 Explain the mechanism for freezing and restarting the process in process migration.

Q. 17 Explain the address space transfer mechanisms in process migration.

Q. 18 Explain message-forwarding mechanism in process migration.

Q. 19 Explain mechanism to handle coprocesses in process migration.

Q. 20 Explain Process Migration in Heterogeneous System.

Q. 21 What are the advantages of process migration? Explain.

Q. 22 Explain the different advantages of threads with compare to process.

Q. 23 Explain different models to construct server process.

Q. 24 Explain different models for organizing threads.

Q. 25 Explain Issues in Designing a Thread Package.

Q. 26 Explain different approaches to implement thread package.

Q. 27 Explain different techniques of thread scheduling.

Q. 28 Explain the role of virtualization in distributed systems.

Q. 29 Explain architecture of virtual machines.

Q. 30 Explain general design issues of server.

Q. 31 Explain the client-side software for distribution transparency.

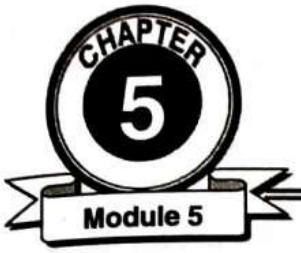
Q. 32 What is server cluster? Explain in brief logical organization of server cluster.

Q. 33 Explain different approaches to code migration.

Q. 34 Explain hoe binding to local resources is handled in process migration.

Q. 35 Write short note on "code migration in heterogeneous systems".

□□□



Consistency, Replication and Fault Tolerance

Syllabus

Introduction to replication and consistency, Data-Centric and Client Centric Consistency Models, Replica Management. Fault Tolerance : Introduction, Process resilience, Reliable client-server and group communication, Recovery.

5.1 Introduction to Replication and Consistency

- The replication of data is carried out in distributed system in order to enhance the reliability and to improve performance. Although reliability and performance can be achieved through replication, it should be ensured to have all the copies of data consistent. If one copy updates then this updates should be propagated to other copies of data also. Otherwise replica will not be same.
- Replication also relates to scalability. If processes access shared data simultaneously the consistency should be ensured. There are many data-centric consistency models. In large scale distributed system, these models are not easy to implement. For this reason, client-centric consistency models are useful, which focus on consistency from the perception of a single client which possibly may be mobile client also.

5.1.1 Reasons for Replication

- Reliability and performance are two main reasons to replicate data. If one replica of the data crashes then other replica can be available for required operation. If single write operation fails in one replica of file out of three replicas then we can consider value returned by other replica. Hence, we can protect our data.
- Replication also improves performance. Scaling of the distributed system can be carried out in numbers or in geographic area. If there is increasing number of processes access the data from same server, then it is better to replicate server and divide the workload. In this way, performance can be improved.
- If we want to scale the system with respect to the size of a geographical area, then replication is also needed. If a copy of data suppose is placed in the proximity of the process which access it, then access time decreases. As a result, the performance as perceived by that process improves.

- Although reliability and performance can be achieved through replication, the multiple copies of the data need to be consistent. If one copy changes then these updates have to be carried out on all copies of data. This price we have to pay against benefits of the replication.

5.1.2 Replication as Scaling Technique

- Replication and caching is mainly used to improve performance. These are also widely applied as scaling techniques. Scalability issues usually come into view in the form of performance problems. If copies of data are placed close to the processes accessing them, then definitely it improves access time and performance as well.
- To have all replicas up to date, updates propagation consumes more network bandwidth. If process P accesses local copy of the data n times per second. Let same copy gets updated m times per second. If n<<m which indicates access-to-update ratio is very low. In this case, many updated versions of data copy will never be accessed by process P. In such case, network communication to update these versions is useless. It requires other approach to update local copy or to not to keep this data copy locally.
- Keeping multiple copies consistent is subject to serious scalability problem. If one process read the data copy then it should always get updated version of data. Hence changes done in one copy should be immediately propagated to all copies (replica) before performing any subsequent operation. Such tight consistency is offered by what is also called synchronous replication.
- Performing update as a single atomic operation on all replicas is difficult as these replicas are widely distributed across large scale network. All replicas should agree on when precisely an update is to be carried out locally. Replicas may need to agree on a global ordering of operations using Lamport timestamps, or this order may be assigned by coordinator. Global synchronization requires lot of communication time, particularly if replicas are spread across a wide-area network. Global synchronization is costly in terms of synchronization.
- The strict assumption about consistency discussed above can be relaxed based on the access and update patterns of the replicated data. It can be agreed that replicas may not be same everywhere. This assumption also depends on purpose for which data are used.

5.2 Data-Centric Consistency Models

- Processes on different machines perform operations on shared data. This shared data is available through use of distributed shared memory, a distributed shared database, or a distributed file system.
- Consider data store which is physically distributed on many machines. Each process has its local copy of the entire data store. The write operation by process changes the data and read operation does not carry out any change in data item. The write operation should be propagated to all other copies of the data store on different machines.
- Read operation by a process on data item should always return the result of last write operation on that data item. A consistency model is basically is a contract between processes and the data store. If processes agree to follow certain rules the data store assures to work correctly. As it is not possible to have global clock, it is difficult to decide exactly which is the last recent write operation. Hence, different model obeys different rules.

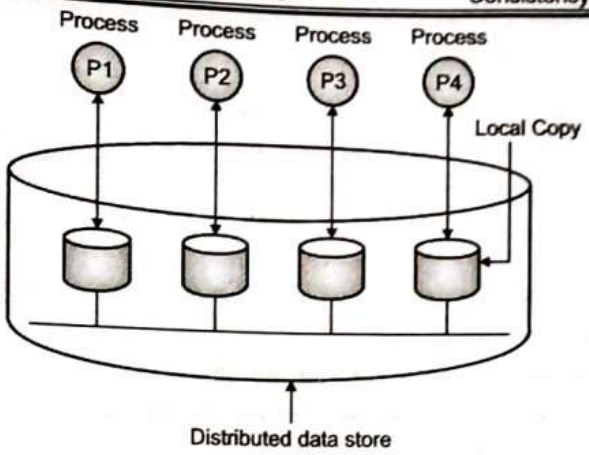


Fig. 5.2.1 : Physically Distributed Data Store

5.2.1 Continuous Consistency

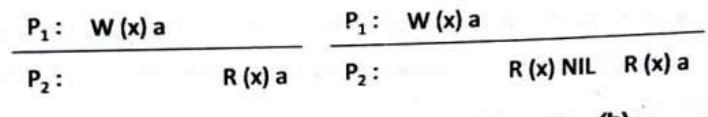
- It is true that, there is no best solution to replicate the data on different machines in network. Efficient solutions can be possible if we consider some relaxation in consistency. The tolerance while considering the relaxation also depends on application. Applications can specify their tolerance of inconsistencies in three ways. These are :
  - o Variation in numerical values between replicas.
  - o Variation in staleness between replicas.
  - o Variation with respect to the ordering of update operations.
- If data have numerical semantics for the applications then they uses numerical deviation or variation. For example, if replicated records contain stock market price data then application may specify that two copies should not deviate more than Rs 10. This would be absolute numeric deviation. On the other hand, a relative numerical deviation could be specified to have deviation between two copies not more than some specific value.
- In both cases of deviations, if a stock goes up without violating the specified numerical deviations, replicas would still be considered to be mutually consistent. A staleness deviation takes in to consideration the last updated time of replica. Some applications can tolerate the data supplied by replica which can be old data, provided it is not too aged. For example, weather reports may remain true for few hours as parameter values do not change suddenly. In such cases, a main server, although it receives current updates, may take decision to propagate updates to the replicas periodically.
- At last, there are some applications that permits ordering of updates that can be different at the different replicas, provided the differences stay bounded. These updates can be viewed as they are applied tentatively to a local copy, until the global agreement from all replicas arrives. As a result, some updates may need to be rolled back and applied in a different order before becoming permanent. Naturally, ordering deviations are not easy to grasp than the other two consistency metrics.

5.2.2 Consistent Ordering of Operations

In parallel and distributed computing, multiple processes share the resources. These processes access these resources simultaneously. The semantics of concurrent access to these resources when resources are replicated led to use of consistency model.

Sequential Consistency

- The time axis is drawn horizontally and interpretation of read and write operation by process on data item is given below. Initially each data item is considered as *NIL*.
  - **$W_i(x)a$:** Process P_i perform write operation on data item x with value a.
  - **$R_i(x)b$:** Process P_i perform read operation on data item x which return value b.
- As shown in following Fig. 5.2.2 (a), process P_1 perform write operation on data item x and modifies its value to a. This write operation is performed by process P_1 on data store which is local to it. We have assumed that data store is replicated on multiple machines. This operation then propagated to other copies of the data store. Process P_2 later reads x from its local copy of the data store and see value a. This is perfectly correct for strictly consistent data store.



(a) (b)

Fig. 5.2.2

- As shown in Fig. 5.2.2 (b), Process P_2 later reads x from its local copy of the data store and see value NIL. After some time, Process P_2 see the value as a. This means it takes some time to propagate the updates from process P_1 to P_2. It is acceptable if we are not considering data store as strictly consistent.
- Sequential consistency model was first defined by Lamport. It is important data-centric consistency model. Following condition data store should satisfy to stay sequentially consistent.
- The result of any execution is the same as if the read and write operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.
- Processes executes concurrently on different machines. Any valid interleaving of read and write operations is tolerable behavior. All these processes see the same interleaving of read and write operations. Time of operation is not considered here.
- Following Fig. 5.2.3 (a) and (b) shows sequentially consistent data store. In Fig. 5.2.3 (a) Process P_1 first performs write on x in its local copy of data store. This sets the value of x as a. Later process P_2 writes b to data item x in its local copy of data store. Both processes P_3 and P_4 first read b and then a. The write operation of process P_2 appears to have occurred before that of process P_1. In Fig. 5.2.3 (b), both processes P_3 and P_4 first read a and then b. Both the data stores shown are sequential consistent.

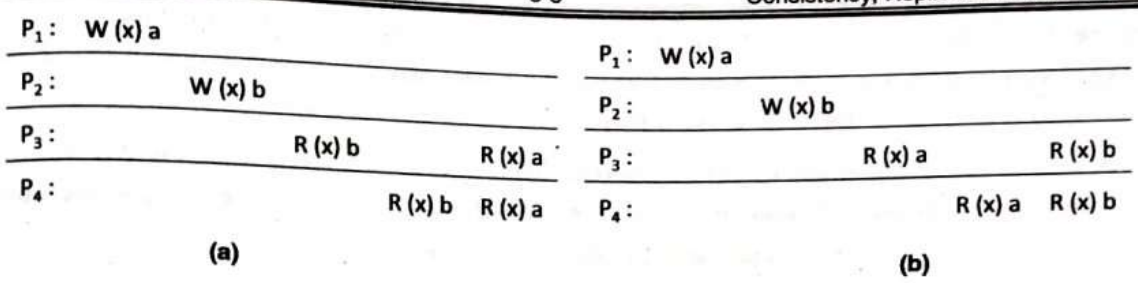


Fig. 5.2.3

- Following Fig. 5.2.4 violets sequentially consistency. Processes P_3 see that first data item been changed to b and later to a. On the other hand, process P_2 will conclude that final value is b.

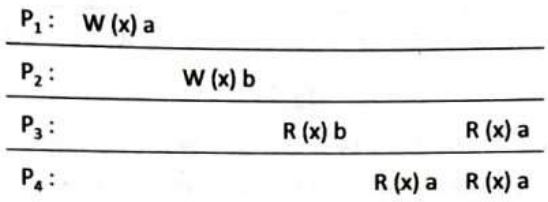


Fig. 5.2.4

- As an example, consider three processes P_1, P_2 and P_3 which are executing concurrently. Three integer type data items a, b, and c are stored in shared sequentially data store. Assignment is considered as write operation and read statement reads two parameters simultaneously. All statement executes indivisibly. Processes are shown below.

Process P_1	Process P_2	Process P_3
a = 1 ;	b = 1 ;	c = 1 ;
read(b, c);	read(a, c);	read(a, b);

Fig. 5.2.5

- Three processes total has 6 statements and hence, 720 (6!) possible execution sequences. If sequence begin with statement a = 1 then total 120 (5!) sequences. The sequences in which statement read(a, c) appears before statement b = 1 and sequences in which statement read(a, b) appears before statement c = 1 are invalid. Hence, only 30 sequences of execution which starts with a = 1 are valid. Similarly 30 starting with b = 1 and other 30 starting with c = 1 sequences are valid. In total, 90 execution sequences are valid out of 720. These can produce different program results which are acceptable under sequentially consistency.

Causal Consistency

- Causal consistency model makes distinction between the events that are causally related and those that are not causally related. If any event b is caused or occurred by previous event y then every process first should see y and then x. Causally consistent data store follows the following condition :
- All processes must see the causally related writes in the same order. Those write which are not causally related (concurrent writes) processes may see in different order on different machines.

- In the Fig. 5.2.6, four processes P_1, P_2, P_3 and P_4 is shown. Process P_1 write on data item x and this value "a" is later read by process P_2. After this, P_2 writes b on data item x. This b value may be the result of computation involving the value read by P_2 which was a. Therefore W(x)a by P_1 and W(x)b by P_2 are causal writes.
- On the other hand, W(x)c by P_1 and W(x)b by P_2 are concurrent writes as computationally not related to each other. Processes P_3 and P_4 may see these writes in different order. It is shown in Fig. 5.2.6. This event sequence is permitted by causal consistent data store but not permitted by sequential consistent data store.

P_1 :	W (x) a			W (x) c	
P_2 :		R (x) a	W (x) b		
P_3 :		R (x) a		R (x) c	R (x) b
P_4 :		R (x) a		R (x) b	R (x) c

Fig. 5.2.6

- In Fig. 5.2.7 (a), W(x)a by process P_1 and W(x)b by P_2 are causally related writes. But, processes P_3 and P_4 see these writes in different order. Hence, it is violation of causally-consistent data store. In Fig. 5.2.7 (b), as W(x)a by process P_1 and W(x)b by P_2 are concurrent writes. This is due to removal of read operation. Hence, figure shows correct sequence of events in a causally consistent data store.

P_1 :	W (x) a			
P_2 :		R (x) a	W (x) b	
P_3 :			R (x) b	R (x) a
P_4 :			R (x) a	R (x) b

(a)

P_1 :	W (x) a		
P_2 :		W (x) b	
P_3 :		R (x) b	R (x) a
P_4 :		R (x) a	R (x) b

(b)

Fig. 5.2.7

FIFO Consistency

- FIFO consistency is the next step in relaxing the consistency by dropping requirement of seeing the causally related writes in same order at all the processes. It is also called the PRAM consistency. It is easy to implement. It says that:
- All the processes must see the writes done by single process in the order in which they were issued. The writes from different processes, processes may see in different order.
- In Fig. 5.2.8, a valid sequence of events for FIFO consistency is shown. Here, W(x)b and W(x)c are the writes from same process P_2. Processes P_3 and P_4 see these writes in the same order i.e. first b and then c.

P_1 :	W (x) a					
P_2 :		R (x) a	W (x) b	W (x) c		
P_3 :					R (x) b	R (x) a	R (x) c
P_4 :					R (x) a	R (x) b	R (x) c

Fig. 5.2.8

Grouping Operations

- Consistency defined at the level of read and write operations does not match its granularity that is offered by applications. The different programs running concurrently use shared data. For this purpose, synchronization mechanisms and transactions are used to control the concurrency between programs. Therefore program level read and write operations are grouped together and bracketed with pair of operations ENTER_CS and EXIT_CS.
- Consider the distributed data store which we have assumed. If process has successfully executed ENTER_CS, it guarantees that its local copy of data store is up to date. Now process can safely execute series of read and write operations inside CS on that store and exit the CS by calling EXIT_CS.
- A series of read and write operations within program are performed on data. This data are protected against simultaneous accesses that would lead to seeing something different than the result of executing the series as a whole. It is needed to have precise semantics about the operations ENTER_CS and EXIT_CS. The synchronization variables are used for this purpose.

Release Consistency

- In release consistency model, acquire operation is used to tell the data store that CS is about to enter and release operation is used to tell that, CS has just been exited. These operations are used to protect the shared data items. The shared data that kept consistent are said to be **protected**. Release consistency guarantees that, when process carry out acquire, the store will make sure that all the local copies of protected data are brought up to date to be consistent with remote copies if must be.
- After release is done, the modified protected data is propagated to other local copies of the store. Carrying out acquire does not guarantee that locally done updates will be propagated immediately to other copies. Carrying out release does not ensure to fetch updates from other copies. Following Fig. 5.2.9 shows valid events for release consistency.

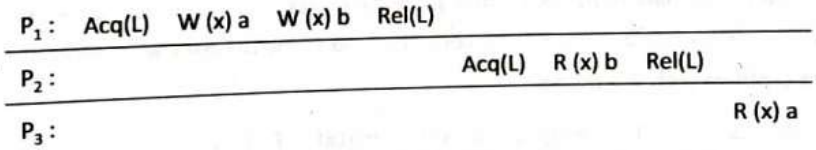


Fig. 5.2.9

- After carrying out acquire, P_1 updates shared data items twice and then carry out release.
- After this, Processes P_2 performs acquire and reads data item x which is b. This value was updated by process P_1 at the time of release. Hence, it is guaranteed that Processes P_2 gets this updated value. On the other hand, process P_3 does not carry out acquire and hence it does not get current value of x. There is no compulsion to get current value of x and returning a is permitted.
- Release consistent distributed data store follows following rules
  - Before carrying out read and write operations on shared data, all previous acquires carried out by process must have completed successfully.
  - Before release is permitted to be carried out, all previous reads and writes carried out by process must have been completed.
  - Accesses to synchronization variables are FIFO consistent.

Entry Consistency

- The shared synchronization variables are used in this model. When process enters the CS, it should acquire the related synchronization variables and when exit the CS, it should release these variables. The current owner of synchronization variable is the process which has last acquired it.
- The owner may enter and exit CSs repetitively without sending any messages on the network. A process not currently having synchronization variable but need to acquire it has to send a message to the current owner asking for ownership and the current values of the data coupled with that synchronization variable. Following rules are needed to follow.
  1. An acquire access of a synchronization variable is not permitted to carry out with respect to a process until all updates to the guarded shared data have been carried out with respect to that process.
  2. Before an exclusive mode access to a synchronization variable by a process is permitted to carry out with respect to that process, no other process may keep the synchronization variable, not even in nonexclusive mode.
  3. After an exclusive mode access to a synchronization variable has been carried out, any other process' next nonexclusive mode access to that synchronization variable may not be carried out until it has carried out with respect to that variable's owner.
- First condition says that at an acquire, all remote updates to the guarded data must be able to be seen. The second condition says that before modification to a shared data item, a process must enter a CS in exclusive mode to confirm that no other process is attempting to update the shared data simultaneously. The third condition says that if a process wants to enter a CS in nonexclusive mode, it must first ensure with the owner of the synchronization variable guarding the CS to obtain the latest copies of the guarded shared data.
- A valid vent sequence for entry consistency is shown Fig. 5.2.10. Lock is associated with each data item. Process P_1 does acquire on x and update it. Later it does acquire on y. Process P_2 does acquire for data item x but not for y. Hence, process P_2 will read value a for data item x but it may get NIL for data item y. Process P_3 does first acquire on y hence it reads value b after y is released by process P_1. The correctly associating the data with synchronization variables is one of the programming problem with this model.

P_1 :	Acq(Lx)	W (x) a	Acq(Ly)	W (y) b	Rel(Lx)	Rel(Ly)	
P_2 :					Acq(Lx)	R (x) a	R (y) NIL
P_3 :						Acq(Ly)	R (y) b

Fig. 5.2.10

Consistency versus Coherence

- Consistency is concerned with the read and writes operations by the processes that are performed on set of data items. A consistency model explains what can be predictable with respect to that set when numerous processes concurrently work on that data.
- Whereas, coherence models expects result from single data item. In this case, assumption is that a data item is replicated at several places; it is said to be coherent when the different copies stick to the policies as defined by its allied coherence model.

5.3 Client-Centric Consistency Models

- In section 5.2, we have considered simultaneous updates on distributed data store. Some distributed data stores may not get updated simultaneously and if, this situation can be easily resolved. On these data store, most of the operations are only reading the data.
- These data stores obeys eventual consistency model which is considered to be weak consistency model. In client-centric consistency models, many inconsistencies can be hidden in a comparatively cheap manner.

5.3.1 Eventual Consistency

- Practically in many situations, concurrency exists in some limited form. If processes only read data from database and hardly some processes performs updates on it then it is not necessary to propagate updates to all processes immediately.
- In World Wide Web (WWW) web pages either can be updated by authority or by owner of that page. In this case, resolving the write-write conflicts at all not require. Conversely, to get better efficiency, browsers and web proxies are frequently configured to hold accessed page in a local cache and to return that page upon the next request.
- Although the web caches returns old pages, it can be acceptable at some extent to requesting client. Eventual consistency states that, updates will be propagated gradually to update the replica if updates do not take place for long time. Eventual consistent data stores work in correct manner, provided clients always access the same replica. But, problems occur when diverse replicas are accessed over a short period of time.
- Suppose, client is mobile client and performing operations on copy of database. Client disconnects with current database and later connects to other replicated copy. In this case, if updates are not propagated from previous copy to recently connected copy of database then he/she may notice inconsistent behavior. Client-centric consistency models resolve such problem. This is typical example of eventual consistency.
- Particularly, client-centric consistency offer assurance *for a single client* about the consistency of accesses to a data store by that client only. No assurance is provided about simultaneous accesses by different clients.
- Consider data store which is physically distributed on many machines. Each process has its local copy of the entire data store. It is assumed that, network connectivity is unreliable. When process accesses data store, it carry out operation on local or nearest copy available. Updates are then eventually propagated to other copies. Following four models are suggested for client-centric consistency.
  - Monotonic Reads
  - Monotonic Writes
  - Read Your Writes
  - Writes Follow Reads
- Following notations are used to describe above models.
  - $x_i[t]$: It indicate the version of data item x at local copy L_i at time t. This version is result of series of write operations at local copy L_i occurred since initialization.
  - $WS(x_i[t])$: Series of write operations on data item x at local copy L_i at time t.
  - $WS(x_i[t_1];x_j[t_2])$: It denote that operations in $WS(x_i[t_1])$ have also been carried out later at local copy L_j at later time t_2.

5.3.2 Monotonic Reads

- Monotonic-read consistency is guaranteed by data store if following condition holds :
- **If a process reads the value of a data item x, any successive read operation on x by that process will always *return that same value or a more recent value.***
- This ensures that in later read operation, process will always get (read) latest or same value (which was returned in last read operation) and not older one. Consider the example of distributed email database. In this case, mailboxes of users are distributed and replicated on multiple machines. The received mail at any location is propagated in lazy manner (as per demand) to other copies of mailbox.
- Only that data gets forwarded which is needed to maintain consistency. Suppose user reads the mailbox in Mumbai and later flies to Delhi. The Monotonic-read consistency guarantees that, the messages in mailbox that were read at Mumbai will also be in mailbox at Delhi.
- Fig. 5.3.1(a) shows monotonic consistent data store. L_1 and L_2 are local copies at different machines. Process P carries out read ($R(x_1)$) operation on x at L_1. The value returned to the process P is the result of write operation $WS(x_1)$ carried out at L_1. Later, process P carries out read operation ($R(x_2)$) on x at L_2. Before P performs read operation on L_2, here writes at L_1 is propagated to L_2 and it is shown in diagram by $WS(x_1; x_2)$. Hence, $WS(x_2)$ is the part of $WS(x_1)$. This shows that monotonic-read consistency is guaranteed.

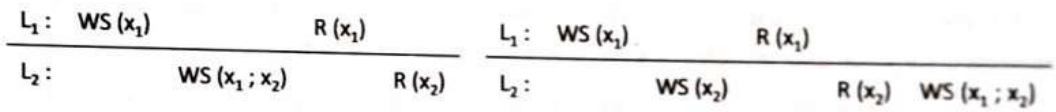


Fig. 5.3.1

- Fig. 5.3.1 (b) shows data store which does not offer monotonic read consistency. After P carries out read ($R(x_1)$) operation on x at L_1, process P carries out read operation ($R(x_2)$) on x at L_2. But only $WS(x_2)$ write operation is performed at L_1. Here, result of writes from L_1 is not propagated to copy L_2. This shows that monotonic-read consistency is not guaranteed in this case.

5.3.3 Monotonic Writes

- In this model, it is important to propagate write operations in the correct order to all copies data store. Monotonic-read consistency is guaranteed by data store if following condition holds :
- **A write operation by a process on a data item x is completed before any successive write operation on x by the same process.**
- This ensures that, result of current write operation by the process on data item reflects the effect of previous write operation by the same process on same data item. The copy of data item may exist anywhere. But updates should be done on copy that is recently updated.
- Consider the updating the software library. In this case, changes are always carried out in one or more functions. With monotonic write consistency promise is given that, whenever changes will be carried out at other copy of the library, all previous updates will be carried out first.

- Monotonic-write consistency model look like data centric FIFO consistency model. Fig. 5.3.2(a) shows monotonic-writes consistency. Process P carries out write ($W(x_1)$) operation on x at L_1. Later, process P carries out write operation ($W(x_2)$) on x at L_2. Here, it is ensured that, previous write operation at L_1 has already been propagated to local copy L_2. This is indicated by operation $WS(x_1)$ at L_2.
- In Fig. 5.3.2(b), monotonic-writes consistency is not guaranteed as write operation at L_1 has not been propagated to copy L_2.

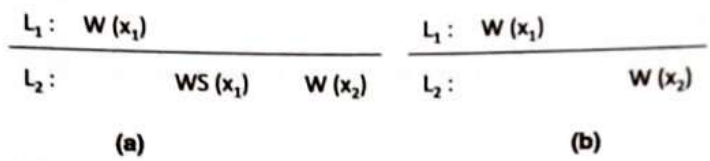


Fig. 5.3.2

5.3.4 Read Your Writes

- Read Your Writes model is closely related to monotonic reads model. Read-your-writes consistency is guaranteed by data store if following condition holds :
- The result of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.
- This ensures that; write operation is always completed before a successive read operation by the same process, regardless of where that read operation occurs.
- Consider the example of changing password of digital library account on web. It takes some time to come in to effect this change. As a result, library may be inaccessible to the user. A reason for delay is use of separate server to manage passwords and it may take some time to subsequently propagate encrypted passwords to the various servers that form the library.
- In Fig. 5.3.3(a), process P carries out write ($W(x_1)$) operation on x at L_1. Later, process P carries out read operation ($R(x_2)$) on x at L_2. Read-your-writes consistency guarantees that, the succeeding read $R(x_2)$ operation at local copy L_2 observes the effects of the write operation $W(x_1)$ carried out at L_1. This is indicated by $WS(x_1; x_2)$, which states that $W(x_1)$ is part of $W(x_2)$.
- Fig. 5.3.3(b) shows that, effect of write operation ($W(x_1)$) on x at L_1 have not been propagated to local copy L_2. This is shown by write operation $WS(x_2)$.

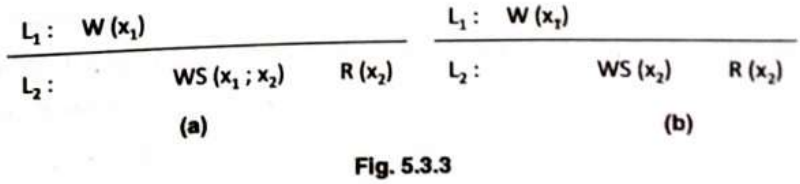


Fig. 5.3.3

5.3.5 Write Follows Reads

- In this consistency model updates of last read operation gets propagated. Write-Follows-Reads consistency is guaranteed by data store if following condition holds:
- A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.

- It states that, process will always perform successive write operation on up to date copy of data item with value most recently read by that process. Consider the example of posting articles and their reactions in network newsgroup. User first read the article X and then posts its reaction Y. As per write-follows-reads consistency, reaction Y will be written to ant copy of the newsgroup only after article X has been written as well.
- In Fig. 5.3.4 (a), process P carries out read ($R(x_1)$) operation on x at L_1. The value x which just now read at L_1, returns value which just written ($WS(x_1)$) on x at L_1. This value also appears in the write set at L_2 where same process later carries out write operation.
- In Fig. 5.3.4 (b), it is not guaranteed that, the operation carried out at L_2 are carried out on a copy that is consistent with the copy just read at L_1.

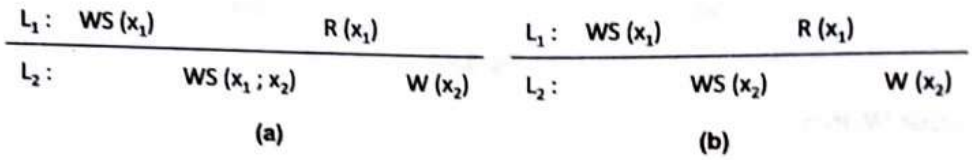


Fig. 5.3.4

5.4 Replica Management

- Replica management is important issue in distributed system. The key issues are to decide where to place replicas, by whom and the time at which placement should be carried out.
- Other issue is selection of approaches to keep the replica consistent. Placement problem also includes placing the servers and placing content. Placing server involves finding the best location and placing content involves which is the best server for placement of content.

5.4.1 Replica-Server Placement

For optimized placement, k best location out of n ($k < n$) needs to be selected. Following are some of the approaches to find best locations.

- Consider distance between clients and locations which is measured in terms of latency or bandwidth. In this solution, one server at a time is selected such that the average distance between that server and its clients is minimal given that previously *k* servers have been already placed (n–k locations are left).
- In second approach, instead of considering the position of clients, take topology of internet formed by autonomous systems (ASs) in which all the nodes runs same routing protocols. Initially take largest AS and place server on router having largest number of network interfaces. Repeat the algorithm with second largest AS and so on. These both algorithms are expensive in terms of computation. It takes $O(n^2)$ time where n are number of locations to be checked.
- In third approach, a region to place the replicas is identified quickly in less time. This identified region comprises collection of nodes accessing the same content, but for which the internode latency is low. The identified region contains more number of nodes with compare to other regions and one of the nodes is selected to play role of replica server.

5.4.2 Content Replication and Placement

Replicas are classified in following three types.

1. Permanent replicas
2. Server-initiated replicas
3. Client-initiated replicas

1. Permanent Replicas

- Initially, distributed data store contains these replicas. These are initial set of replicas that represents the distributed data store. These replicas are less in number in many cases. For example, in distribution of web site, the files that form the site are distributed across few servers in single location. The incoming request for file gets forwarded to one of the server, may be in round robin manner.
- In other type of distribution mirroring is used in which case, a web site is copied to few number of servers called as mirror sites. These servers are geographically spread across internet. The clients request gets forwarded to one of the server. Mirroring approach is used with cluster-based Web sites having limited number of replicas, which are more or less statically configured. As another example, a distributed database can be distributed and replicated to number of servers that collectively form a cluster of servers.

2. Server-Initiated Replicas

- These replicas are created by the owner of the data store in order to enhance the performance. Suppose Web server placed in a city easily handles incoming requests and over a couple of days, sudden burst of requests arrive to this server from unexpected location which is far from server. It may be useful to place the temporary replicas in regions from where requests are coming.
- Web hosting services offers collection servers placed across the internet. These servers stores and offers access to the web files which belongs to third parties. These hosting services can dynamically replicates files to the server where it is needed in order to improve performance. If servers are already placed then content placement is easier than placement of servers.
- The algorithm supports web pages and it is considered that updates are comparatively less than read requests. Two issues are considered by the algorithm. First, it reduces load on server. Second, the needed files can be migrated from current server to the servers that is available in the proximity of clients that issue many requests for those files.
- In this algorithm, each server keeps track on count of number of accesses to the file by clients and the locations of the incoming requests. For a particular client C, any server involved in web hosting service can determine closest server to this client. This information, servers may take from routing database.
- If file F is stored at sever S_2 and sever S_1 is closest to the clients C_1 and C_2. In this scenario, all access requests for file F at server S_2 from C_1 and C_2 are together gets registered at S_2 as a single count of accesses $CS_2(S_1, F)$. A deletion threshold for file F at server S *(del(S, F))* is also maintained. It says that, if count of incoming requests from clients for file F at server S drops below this threshold then file should be removed from server S.
- Removing the file from server in this manner again reduces number of replicas. This can again increase the load on other servers. So, some special actions are carried out in order to survive at least one copy of the same file.

- In the same way, file F is replicated if incoming requests for it exceeds the replication count *rep(S, F)*. If count of requests for file F is in between these two thresholds then file is permitted to migrate on server in proximity of clients requesting it.
- Every server again assesses the placement of file which it stores. In case of number of access requests count if drops below threshold *(del(S, F))* at server S, it deletes F, provided it is not a last copy. Certainly, replication is carried out only if the total number of access requests for *F* at S exceeds the replication threshold *rep(S, F)*.

3. Client-Initiated Replicas

- These replicas are initiated by client and also called as client caches. Caches are always said to be managed by clients. In many cases, the cached data should be inconsistent with data store. Client caches improve the access time to data as accesses are carried out from local cache. This local cache could be maintained on client's machine or in separate machine in same LAN as client. Cache hit occurs if data is accessed from the cache.
- Cache holds the data for short period. This is either to prevent extremely stale data from being used or to make room for other requested data from data store. Caches can be shared between many clients to improve cache hits. The assumption is that, requested data by one client may be useful to another nearby client. This assumption may be correct for some types of data stores.
- A cache either is usually placed on the same machine as its client or on a machine shared by clients on the same LAN. In some other cases, system administrators may place a shared cache between a number of departments, organizations, or for whole country. Other approach is to place servers as caches at some specific locations in wide area network. The client locates nearest server and requests it to keep copies of the data the client was previously accessing from somewhere else.

5.4.3 Content Distribution

Management of the replicas also involves propagation of updated contents to the appropriate servers.

State versus Operations

Following are the different possibilities for propagation of updates.

- Only a notification of an update is propagated.
- Data is transferred from one copy to other copy.
- Update operation is propagated to other copies.

Propagation of notification of update

- In an invalidation protocol, if updates occur in one copy then other copies are informed about these update. This informs to other copies that the data they hold are no longer valid. It may specify that, which part of the data store has been updated, so that only changed part of a copy is actually invalidated.
- In this case, only notification is propagated. Before carrying out the update operation on an invalidated copy, it needs to be updated first, depending on the particular consistency model that is to be supported. The network bandwidth needed to propagate notifications is less. Invalidation protocol works best in situation where many update operations are carried out compared to read operations (read-to-write ratio is small).

- As an example, consider the data store where updates are propagated by transferring modified data to all other copies. Suppose, frequent updates takes place compared to read operation. Consider two consecutive update operations. Since there is no read operation has been performed between two updates, second update will simply overwrite the first update in all copies. In this case sending notification only will be more beneficial.

Propagation of updates by transferring the data

- This approach is more suited when read-to-write ratio is relatively high. In this case, chances are more of modified data will be read is before next update takes place. Hence, in such situation transferring the modified data to all other copies is necessary.
- In order to save the bandwidth, log of changes made can be sent instead of sending the modified data. Communication overhead can also be saved by sending single message for multiple modifications. In this case, these modifications are packed in to single message.

Propagation of update operations

- This approach informs to each copy about which update operation it should carry out. It sends only parameter values which will be required to carry out update operation at target copies. If size of parameters to be sent is relatively small then propagating update operations requires less bandwidth.
- It is also called as active replication. At each replica more processing power is required if operations to be carried out are more complex.

Pull versus Push Protocols

- Push Protocols are server-based protocols in which updates are propagated by servers themselves to the replicas although they do not request for these updates. Push-based approaches usually used between permanent replicas server-initiated replicas, but can also be used to send updates to client caches. Push Protocols are generally used where maintaining high degree of consistency is necessary.
- The largely shared replicas, permanent replica and server-oriented replicas are often shared between many clients. Clients frequently carry out read operation on these replicas. In this case, read-to-update ratio at each replica is comparatively high. Hence, each read operation by clients expected to get updated copies. If client asks for recently updated copy then push-based protocol immediately makes it available.
- On the other hand, pull-based protocols works to get updates as per request issued. In this case, either server or client sends request to another server to obtain any updates done recently there. These protocols are called as client based protocol. For example, In case of web caches whenever requests arrives for the data, they often check with original web server to confirm whether cached data is up to date or not since they were cached.
- If cached data are modified at original web server then it gets transferred to cache and then returned to the requesting client. A pull-based protocol is efficient when update operations are relatively high compared to read operations (read-to-update ration high). This is when there are non shared client caches. In this case there is single client. If caches are shared among many clients then also this protocol is efficient. The main disadvantage of a pull-based approach compared to a push based approach is increase in response time in case of cache miss.

- Consider the situation with single (non-distributed) server and many clients :
  - In this case, push base protocol has to keep track of many client caches. This puts load on server. Server should be stateful server although it is less fault tolerant. In pull-based approach, server does not require to keep track of many client caches.
  - In push-based approach, server sends the messages to clients. These are update or fetch update later messages. In pull-based approach, client sends the messages to server. Client has to poll the server to check if any updates occurred and if then fetch the updates.
  - When a server sends modified data to the client caches, the response time at the client side is zero. When server pushes invalidations, the response time is the same as in the pull-based approach, and is decided by the time involved in accessing the modified data from the server.
- As a balance between push and pull-based approaches, a hybrid form of update propagation based on leases is introduced. A lease is a time interval in which server pushes updates to clients. In other word, it is a promise by the server that it will push updates to the client for a specific time. After expiry of the lease, the client is forced to poll the server for updates and pull in the modified data if required. In other approach, a client requests a new lease to push updates when the earlier lease expires.

Unicasting versus Multicasting

- The use of unicasting or multicasting depends on whether updates needs to be pushed or pulled. If server is a part of data store and push updates to other n number of servers then it sends n separate messages. In this case, with multicasting underlying network takes care of sending n messages.
- Suppose all replicas are placed in LAN then hardware broad casting is available. In that case broadcasting or multicasting is cheaper. In this case, unicasting the update is expensive and less efficient. Multicasting can be combined with push based approach where single server sends message to multicast group of other n servers.

5.5 Fault Tolerance

In distributed system, partial failure may occur. In partial failure, failure of one component of system may affect operation of other component but yet some other components may remain unaffected. The distributed system should be designed to recover from such partial failure automatically without affecting the performance.

5.5.1 Basic Concepts

- Distributed system should tolerate faults. Fault tolerant system is strongly related to dependable system. Dependability covers following requirements for distributed system along with other important requirements.
  - **Availability :** It refers to the probability that system is performing its operation correctly. If system is working at any given instant of time then it is highly available system.
  - **Reliability :** It refers to the working of system continuously without failure or interruption during a relatively long period of time. Although system may be continuously available, but we cannot say that it is reliable.
  - **Safety :** Safety refers to the situation that when a system temporarily fails to function correctly, nothing catastrophic takes place.
  - **Maintainability :** It refers to how failed system can be repaired without any trouble. System should be easily maintainable which will then show high degree of availability.

- Dependable system should also offer high degree of security. The system is said to fail if it is not offering all the services for which it was designed. An error is a part of a system's state that may cause a failure. The cause of an error is called a fault. System should provide its services although faults exist in the system.
- There are three types of faults. These are transient, intermittent, and permanent. Transient faults occur once and then vanish. If the operation is repeated, the fault goes away. An intermittent fault occurs, then disappears of its own accord, then again appears, and so on. A permanent fault continues to live until the faulty part is replaced.

5.5.2 Failure Models

- In distributed system, communication channel, processes, machines can fail during the course of execution. Failure model suggest the ways in which failures in different components of the system may occurs and to provide the understanding of this failure.
- Suppose in distributed system, servers communicate with each other and with their clients. This system is not satisfactorily offering services indicates that servers, communication channels, or possibly both, are not carrying out the work what they are supposed to do.
- In this example, **crash failure** occurs when server halt but it was working correctly until it halts. If server fails to give reply to the incoming requests from other servers or clients then it is considered to be **omission failure**. If server fails to send the message then it is **send omission failure** and if fails to receive the incoming message then it is **receive omission failure**. If server response is not within the specified time interval then it is **timing failure**. If server gives incorrect response then it is **response failure**. **Value failure** occurs when value of the response is wrong. If server deviates from the correct flow of control then it is **state transition failure**. Suppose server produces arbitrary responses at arbitrary times then it is referred as **arbitrary failure**.

Omission Failures

The omission failure refers to the faults that occur due to failure of process or communication channel. Because of this failure, process or communication channel fails to carry out the action or task that it is supposed to do.

Process Omission Failures

- The main omission failure of process is when it crashes and never executes its further action or program step. In this case, process completely stops. When any process does not responds to requesting process repeatedly, the requesting process detects or concludes this crash.
- The detection of crash in above manner relies on use of timeouts. In asynchronous system, timeout can point towards only that process is not responding. The process may have crashed, may be executing slowly or messages have yet not delivered to the system.
- If other process surely detects the crash of process then this process crash is called as fail-stop. This fail-stop behaviour can be formed in synchronous system when processes uses time out to know when other process fail to respond and delivery of messages are guaranteed.

Communication Omission Failures

- Sending process P executes *send* primitive and puts message in its outgoing buffer. Communication channel transport this message to receiving process Q's incoming buffer. Process Q then executes *receive* primitive to take this message from its incoming buffer. It then delivers the message.

- If communication channel is not able to transport message from process P's outgoing buffer to process Q's its incoming buffer then it produces omission failure. The message may be dropped due to non-availability of buffer space at receiving side, no buffer space at intermediate machine or network error detected by checksum calculation with data in message.
- Send-omission failures refer to loss of messages between sending process and its outgoing buffer. Receive- omission failures refer to loss of messages between incoming buffer and the receiving process.

Arbitrary Failures

- In this type of failure process or communication channel behaves arbitrarily. In this failure, the responding process may return wrong values or it may set wrong value in data item. Process may arbitrarily omit intentional processing step or carry out unintentional processing step.
- Arbitrary failure also occurs with respect to communication channels. The examples of these failures are: message content may change, repeated delivery of the same messages or non-existent messages may be delivered. These failures occur rarely and can be recognized by communication software.

Timing Failures

- In synchronous distributed system, limits are set on process execution time, message delivery time and clock drift rate. Hence, timing failures are applicable to this system.
- In timing failure, clock failure affects process as its local clock may drift from perfect time or may exceeds bound on its rate.
- Performance failure affects process if it exceeds the defined bounds on the interval between two steps.
- Performance failure also affects communication channels if transmission of message take longer time than defined bound.

Masking Failures

- Distributed system is collection of many components and components are constructed from collection of other components. Reliable services can be constructed from the components which exhibit failures.
- For example, suppose data is replicated on several servers. In this case, if one server fails then other servers would provide the service. Service mask failure either by hiding it or by transforming it in more acceptable type of failure.

5.5.3 Failure Masking by Redundancy

- The use of redundancy is main technique for masking the faults. Redundancy is categorized as information redundancy, time redundancy, and physical redundancy. With information redundancy extra bits are added for recovery. For example, hamming code added at sender side in transmitted data for recovery from noise.
- Time redundancy is particularly useful in case of transient or intermittent faults. In this, action is performed again and again if needed with no harm. For example, aborted transaction can be redone with no harm.
- With physical redundancy, to tolerate the loss or malfunctioning of some components either extra hardware or software components are added in system.

5.6 Process Resilience

In distributed system, failure of processes may happen. In this case, fault tolerance is achieved by replicating processes into groups.

5.6.1 Design Issues

- If identical processes are organized in group, then message sent to this group gets delivered to all the group members. Every process in group receives this message. If one of the members fails the other can take over this process's job.
- These process groups may be managed dynamically. It allows creating new groups and destroying old groups. During the system operation, a process can join a group or leave it. A process can be a member of multiple groups simultaneously. As a result, mechanisms are required for group management and group membership.
- Groups permit processes to deal with sets of processes as a single abstraction. Hence a process can send a message to a group of servers without knowing to which servers or number of servers and their locations, which may vary from one call to the next.
- Two types of groups exist. In **flat group**, all the processes are at equal level and all decisions are made collectively. The advantage of this organization is that there is no single point of failure. But decision making is complex as all are involved in this process. Voting incur delay and overhead.
- In simple **hierarchical group**, one process acts as coordinator and all the others are workers. In this group, a request from worker or external client is first gets handed over to coordinator and then it decides which worker is appropriate to carry it out, and forwards it there. In this group single point of failure is the problem. If coordinator crashes the whole group will affect.

Group Membership

- A group server is responsible to create and destroy the groups. It also allows the process to join or leave the group. The group server maintains a complete data base of all the groups and their exact membership. In this, single point of failure is again the problem. If group server crashes then group management will affect.
- Another approach is to manage membership in distributed manner. if reliable multicasting is existing, an external process can send a message to all group members declaring its wish to join the group. To leave a group, a member just sends a goodbye message to every other member.
- As soon as process joins the group, it should receive all the messages sent to that group. After leaving group, process should not receive any message from that group and any other group members should not receive messages from it. There should be some protocol to deal with the situation where many machine suppose fails and group can no longer function at all.

5.6.2 Failure Masking and Replication

- Group of identical processes permits us to mask one or more faulty processes in that group. A group can be formed by replicating the processes and organizing then in a group. Hence, single vulnerable process can be replaced by this group to tolerate the fault. Replication can be carried out either with primary-based protocols, or through replicated-write protocols.

- Primary-based replication for fault tolerance usually emerges in the form of a primary-backup protocol. In this case, a process group is organized in a hierarchical manner in which a primary coordinates all write operations. Primary is fixed but its role can be taken over by backups whenever needed. In a situation of crash of primary, the backups run some election algorithm to elect a new primary.
- Replicated-write protocols are used in the form of active replication, in addition to using quorum-based protocols. These solutions consider the flat group. The main advantage is that such groups have no single point of failure, at the cost of distributed coordination. The advantage of this organization is that there is no single point of failure. But distributed coordination incurs delay and overhead.
- Consider the replicated write systems. Amount of replication required is an important issue to consider. A k fault tolerant system survives faults in k components and still meets its specifications. If processes fail slowly, then having k + 1 of them is sufficient to give k fault tolerance. If k of them just stops, then the reply from the other one can be used. In case of Byzantine failures, processes continuing to run when sick and sending out erroneous or random replies, a minimum of 2k +1 processors are required to attain k fault tolerance.

5.6.3 Agreement in Faulty Systems

- Fault tolerance can be achieved by replicating processes in groups. In many cases, it is required that, process groups reaches some agreement. For example; electing a coordinator, deciding whether or not to commit a transaction, dividing up tasks among workers etc. Reaching such agreement is uncomplicated, provided communication and processes are all perfect. Otherwise problem may arise.
- The general aim of distributed agreement algorithms is to have all the correctly working processes reach an agreement on some issue, and to build that consensus within a finite number of steps. Following cases should be considered.
  - Synchronous versus asynchronous systems: A system is synchronous if and only if the processes are known to operate in a lock-step mode.
  - Communication delay is bounded or not: Bounded delay means every message is delivered with a globally and predetermined maximum time.
  - Message delivery is ordered or not: Messages from same sender is delivered in the same order in which they were sent.
  - Message transmission through Unicasting or Multicasting.

Byzantine Agreement Problem

- In this case, we assume that processes are synchronous, messages are unicast while preserving ordering, and communication delay is bounded. Consider n number of processes. In this example, let n = 4 and k = 1 where k is number of faulty processes. The goal is that, each process should build the vector of length n.
- **Step 1 :** Each non-faulty process i send v_i to other process. Reliable multicasting is used to send the v_i. Consider Fig. 5.6.1. In this case, process 1 send 1, process 2 send 2 and process 3 lie to everyone, sending x, y, and z respectively. Process 4 send 4. Process 3 is faulty process and process 1,2 and 4 are non faulty processes.

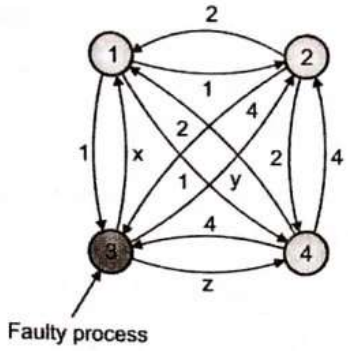


Fig. 5.6.1

- **Step 2 :** The results collected by each process from other processes in the form of vectors is shown below.
  - Process 1: (1, 2, x, 4)
  - Process 2: (1, 2, y, 4)
  - Process 3: (1, 2, 3, 4)
  - Process 4: (1, 2, z, 4)
- **Step 3 :** Now, every process passes its vector to ever other process. The vectors that each process receives from other each process are shown below. Since process 3 is faulty process, it lies and sends new values a to l. these are new 12 values.

Process 1	Process 2	Process 4
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

- **Step 4 :** Now, each process inspects the i^{th} element of each of the newly received vectors. If any value has a majority, that value is put in **Result** vector. If no value has a majority, the corresponding element of the result vector is marked *UNKNOWN*. Processes 1, 2, and 4 all come to agreement on the values for v_1, v_2, and v_4, which is the correct result.

  Result = (1, 2, *UNKNOWN*, 4)
- These processes conclude that about v_3 cannot be decided, but is also irrelevant. The goal of Byzantine agreement is that consensus is reached on the value for the non-faulty processes only.
- If n = 3 and k=1 then only two non-faulty processes and one faulty process. It is shown in Fig. 5.6.2.

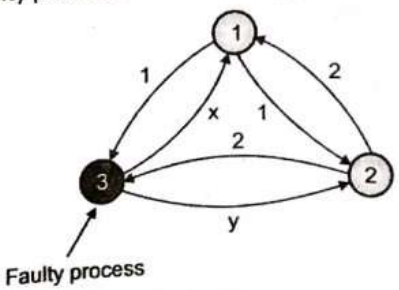


Fig. 5.6.2

- The results collected by each process from other processes in the form of vectors are shown below.
  - Process 1: (1, 2, x)
  - Process 2: (1, 2, y)
  - Process 3: (1, 2, 3)
- Following vectors are received by processes. There is no majority for element 1, element 2 and 3 also. All will be marked as *UNKNOWN* and hence, algorithm is failed.

Process 1	Process 2
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

- In system, if k faulty processes are there then agreement can be achieved if 2k+1 correctly functioning (non-faulty) processes present for a total of 3k+1.

5.6.4 Failure Detection

- For proper masking of failures, it is necessary and requirement to detect them. It is needed to detect faulty member by non-faulty processes. The failure of member detection is main focus here. Two approaches are available for detection of process failure. Either process asks by sending messages to other processes to know whether others are active or not or inactively wait until messages arrive from different processes.
- The latter approach useful only when it can be certain that there is sufficient communication between processes. In fact, actively **pinging** processes is generally followed. The suggested timeout approach to know whether a process has failed or not, suffers with two major problems. As a first problem, declaring process's failure as it does not return response to ping messages is wrong as this may due to unreliable networks. In other argument, there is no practical implementation to correctly determine failure of process due to timeouts.
- Gossiping between processes can also be used to detect faulty process. In gossiping, process tells other process about its active state. In other solution, neighbors probe each other for their active state instead of relying on single node to take decision. It is also important to distinguish the failure due to process and underlying network.
- This problem can be solved by taking feedback from other nodes or processes. In fact, when observing a timeout on a ping message, a node requests other neighbors to know whether they can reach the presumed failing node. If a node is still active, that information can be forwarded to other interested nodes.

5.7 Reliable Client-Server Communication

- While considering the faulty processes, fault tolerance in distributed system should also consider the failures of the communication channels. Communication failures are also equally important in fault tolerance. Communication channels may also suffer through crash, omission, timing, and arbitrary failures.
- Reliable communication channels are required for exchanging the messages. Practically, while designing the reliable communication channels, main focus is given on masking crash and omission failures. Arbitrary failure may occur in the form of duplicate messages due to retransmission by original sender.

5.7.1 Point-to-Point Communication

- TCP is reliable transport protocol which supports reliable point-to-point communication in distributed system. TCP handles lost of messages is by using acknowledgement and retransmission. Hence, this omission failure is masked by TCP. Client cannot notice such failures.
- If TCP connection is suddenly broken then it is crash failure. In this failure, no more messages can be transmitted through the communication channel. This failure is only masked through establishing new connection by distributed system.

5.7.2 RPC Semantics in the Presence of Failures

- In Remote procedure calls (RPC), communication is established between the client and server. In RPC, a client side process calls a procedure implemented on a remote machine (server). The primary goal of RPC is to hide communication while calling remote procedure so that remote procedure calls looks like just local procedure calls.
- If any errors occur, then masking differences between remote and local procedure calls is not easy. In RPC system, following failure may occur.
  - The client cannot locate the server.
  - The request message from the client to the server is lost.
  - The server crashes after receiving a request.
  - The reply message from the server to the client is lost.
  - The client crashes after sending a request.

The Client Cannot Locate the Server

- In this case, it may happen that all servers are down. In other case, version of client stub is old. Meanwhile server evolves and new interfaces are installed and compiled. At server side new version of stub is present. Hence, binder will be unable to bind with server and failure will be reported.
- As a solution to this problem, an exception can be raised upon errors. Many language supports for writing procedures that are invoked upon errors. Signal handler can also be used for this purpose. Again drawback is that, many languages do not supports exceptions or signals. It is not possible to achieve transparency by using exceptions or signal handlers.

Lost Request Messages

- To deal with this problem, OS or client stub starts timer while sending request. The retransmission of request is carried out if acknowledgement or reply does not arrive before timer expires. Server will not be able to differentiate between original and retransmission if message was truly lost. In this case, no problems arise.
- Suppose so many client request messages are lost. In this case, client will falsely conclude that server is down. Then again situation will be like "cannot locate the server". If request message was not lost then let the server handle it.

Server Crashes

- Following sequence of events takes place when client sends request to server.
  1. A request arrives at server. Server has processed the request and reply is sent to client.

2. A request arrives at server. Server has processed the request. Server crashes before it can send the request.
3. A request arrives at server. Server is crashed before processing the request.

- In case (2) and (3) different solutions are required. In (2), system has to report failure to the client. In (3) retransmission of request is required. Client retransmits when its timer expires.
- In (3), Client wait till server reboots. Client may rebind to new server. At **least one semantics** technique can be used. It means, try until reply arrives from server. It means, RPC has been carried out at least one time, but possibly more.
- In **at-most-once semantics** technique, failure is immediately gets reported to the client. It guarantees that the RPC has been carried out at most one time, but possibly none at all. In other technique, client gets no help and no guarantee. RPC has been carried out from 0 to any number of times. It is easy to implement.

Lost Reply Messages

- In this case client retransmits request message when reply does not arrive before timer expires. It concludes that request is lost and then retransmits it. But, in this case client actually remains unaware about what is happened actually. Idempotent operations can be repeated. These operations produce same result although repeated any number of times. Requesting first 512 bytes of file is idempotent. It will only overwrite at client side.
- Some request messages are non-idempotent. Suppose, request to server is to transfer amount from one account to other account. In this case, each retransmission of the request will be processed by server. Transfer of amount from one account to other account will be carried out many times. The solution to this problem is, try to configure all requests in idempotent way. This is not possible as many requests are inherently non-idempotent.
- In another solution, client assigns sequence number to each request. Server can then keep track on most recent request and it can differentiate between original (first) and retransmitted requests. Now server can reject to process any request a second time. Client may put additional bit in first message header so that server will take it for processing. This is required as retransmitted message requires more care to handle.

Client Crashes

- In this case, client sends request to server and before reply arrives it crashes. The computation of this request is now going on at server. Such computation for which no one is waiting is treated as an **orphan.** This computation at server side (orphan) wastes CPU cycles, locks files, hold resources.
- Suppose client reboots and sends same request. If reply from orphan arrives immediately then there will be confusion state. This orphan needs solution in RPC operation. Four possible solutions are suggested as below.

1. **Orphan Extermination :** Client maintains log of message on hard disk mentioning what it is about to do. After a reboot, log is verified by client and the orphan is explicitly killed off. This solution needs to write every RPC record on disk. In addition, these orphans may also do RPC, hence, creating grandorphans or further descendents that are difficult or practically not possible to locate.
2. **Reincarnation :** In this solution, no need to write log on disk. Time is divided in sequentially numbered epochs. Client broadcasts a message to all machines telling the start of a new epoch after its reboot. Receivers of this broadcast message then kills all remote computations going on the behalf of this client. This solution also suffers through network delay and some orphan may stay alive. Fortunately, however, when they report back, their replies will include an out of date epoch number so that they can easily notice it.

3. **Gentle Reincarnation :** When an epoch broadcast arrives, each machine checks about any remote computations running locally, if running, tries its best to locate their owners. Computations are killed in case owners cannot be located.

4. **Expiration :** In this solution, RPC is supposed to finish the job in specified time quantum T. If not finishes in this time, another time quantum is requested which is a nuisance. In contrast, if after a crash the client waits a time T prior to rebooting, all orphans are certain to be gone.

5.8 Reliable Group Communication

5.8.1 Basic Reliable-Multicasting Schemes

- Reliable multicast services promise that messages are delivered to all members in a process group. Most of the transport layers only offer point-to-point connection between two processes. They seldom offer reliable communication to a collection of processes. So each process has to set up reliable point-to-point connection with other process. If number of processes is large then achieving reliability through point-to-point connection is difficult. If number of processes is small then it is simple to achieve reliability through point-to-point connection.
- All the processes in group should receive incoming message for that group. In other word, message sent to particular process group should be delivered to the all members of that group. There are some situations that need to be considered. These are: if process joins the group during communication, whether this newly joined process should receive the message or not and if a sending process crashes during communication.
- It is necessary to differentiate between reliable communication in the presence of faulty processes and reliable communication when processes functions correctly. In case of reliable communication in the presence of faulty processes, multicasting is said to be reliable if it is guaranteed that, all non-faulty processes in group receives message. It is necessary to arrive at an agreement on what the group actually looks like before a message can be delivered, besides various ordering constraints.
- It is simple to handle the case when already agreement exists on who is a member of the group and who is not. Especially, if it is assumed that, processes do not fail, and processes do not join or leave the group during communication then reliable multicasting simply means that every message should be delivered to each current group member.
- Sometimes, it is requirement of receiving the messages by all group members in same order. In simple case, group members can receive messages in any order and it is easy to implement if number of receivers are small.
- Consider underlying communication system offers only unreliable multicasting. In this case, some messages from sender may be lost or may not be delivered to all the receivers (group members). In this case, a sequence number is assigned by sender to each message it multicasts. Suppose messages are received in the order they are sent. Sender maintains buffer to store already sent messages till acknowledgment (ACK) for that message is received. It is easy for a receiver to detect it is missing a message. For missed message by receiver, sender receives negative ACK (NACK). Sender then retransmits the message. In other solution, sender can retransmits message when it has not received all acknowledgments within a certain time. Piggybacking can be used to minimize number of messages.

5.8.2 Scalability in Reliable Multicasting

- Above reliable multicast scheme cannot support large numbers of receivers. Sender has to receive n ACKs from n receivers. If large numbers of receivers are there then the sender may be swamped with such feedback messages called as feedback implosion. Also, the receivers are spread across a wide-area network.
- Instead of sending ACKs by receivers, only NACKs can be sent to sender. Hence, better scaling can be achieved by reducing number of feedback messages. In this case, there will be less chances of feedback implosion. In this case also sender has to keep messages in history buffer to retransmit the message which is not delivered. Sender has to remove messages from history buffer after some time has elapsed in order to prevent it from overflowing. This may cause problem if NACK arrives and same message is already removed from the buffer.

Nonhierarchical Feedback Control

- It is necessary to reduce number of feedback messages while offering the scalable solution to reliable multicasting. A more accepted model used in case of several wide-area applications is **feedback suppression.** This approach underlies the Scalable Reliable Multicasting (SRM) protocol. In SRM, receiver only sends NACK for missed message. For successfully delivered messages, receiver does not report acknowledgements (ACKs) to sender. Only NACKs are returned as feedback. Whenever a receiver notices that it missed a message, it multicasts its feedback to the rest of the group.
- In this way, all the members of the group know that message m is missed by this receiver. If k number of receivers has already missed this message m, then each of k members has to send NACK to sender so that m can be retransmitted. On the other hand, suppose retransmissions are always multicast to the entire group, only a single request for retransmission can be sent to sender.
- For this reason, a receiver R within group who has missed the message m sends the request for retransmission after some random time has elapsed. If, in the meantime, another request for retransmission for m reaches R, R will hold back its own feedback (NACK), knowing that m will be retransmitted soon. In this manner, only a single feedback message (NACK) will reach sender S, which in turn next retransmits m.
- Feedback suppression has been used as the fundamental approach for a number of collaborative Internet applications. Although, this mechanism has shown to scale reasonably well, it also introduces a number of serious problems. It requires accurate scheduling of feedback messages at each receiver so that single request for retransmission will be returned to the sender. It may happen that, many receivers will still return their feedback simultaneously. Setting timers for that reason in a group of processes that is spread across a wide-area network is difficult.
- Other problem in this mechanism is that, a retransmitted message also gets delivered to the group members who have already successfully received this message. Unnecessarily, these processes have to again process this message. A solution to this problem is to let these processes which have missed the message m, join another multicast group to receive message m. This solution requires efficient group management which is practically not possible in wide area network. A better approach is therefore to let receivers that tend to miss the same messages team up and share the same multicast channel for feedback messages and retransmissions.

- The scalability of SRM can be improved, if receivers of message m assist in local recovery. In this solution, a receiver of message m receives a request for retransmission, it can make a decision to multicast m even before the retransmission request reaches the original sender.

Hierarchical Feedback Control

- For large groups of receivers, hierarchical feedback control is better approach.

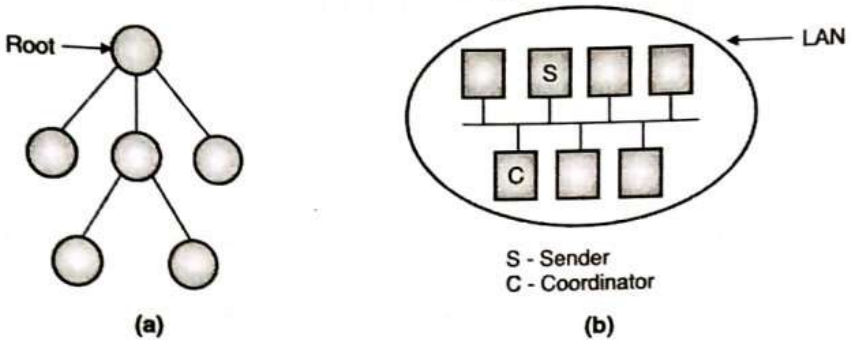


Fig. 5.8.1

- Consider only a single sender multicast messages to a very large group of receivers. The group of receivers is partitioned into a number of subgroups. These subgroups then organized into a tree. The root of the tree is formed by the subgroup in which sender is present. Any reliable multicasting scheme appropriate for small groups can be used within any subgroup. The tree is shown in Fig. 5.8.1 (a). Each node in the tree represents subgroup which has organization as shown in Fig. 5.8.1 (b).
- Each subgroup appoints local coordinator C which handles retransmission requests of the members (receivers) of the same subgroup. The local coordinator also maintains its own history buffer. If the coordinator itself has missed a message m, it requests the coordinator of the parent subgroup to retransmit message m. In a scheme based on ACKs, a local coordinator sends an ACK to its parent if it has received the message. If a coordinator has received ACKs for message m from all members in its subgroup, as well as from its children, it can remove m from its history buffer.
- The construction of the tree is main difficulty in this approach. In many cases, it is necessary to build tree dynamically. One technique is to use the multicast tree in the underlying network, if there is one. In principle, the approach is then to improve each multicast router in the network layer so that it can proceed as a local coordinator in the manner just described. Practically, such adaptations to existing computer networks are difficult to carry out. As a result, application-level multicasting solutions are popular.
- Finally, designing reliable multicast schemes that are scalable to a large number of receivers spread across a wide-area network is a complex problem. No single best solution is available, and each solution introduces new problems.

5.8.3 Atomic Multicast

- Now we consider reliable multicasting in the presence of process failures. In particular, distributed system should guarantee that a message is delivered to either all processes or to none at all. It is often needed in many situations. Also, it is also necessary requirement that that all messages are delivered in the same order to all processes called as the atomic multicast problem.

- Consider a replicated database constructed as an application on top of a distributed system. Suppose distributed system offers reliable multicasting facilities. Hence, it permits construction of process groups to which messages can be reliably sent. The replicated database also forms process group. There is one process for each replica of the database. All these processes belong to one group.
- The update operations are multicast to all replica and operations are carried out locally. A series of update operations is to be carried out on database. If one of the replica crashes during one update operation then this update is lost for this replica. Whereas, same update is correctly carried out at the other replicas.
- The crashed replica can recover to the same state it had before the crash when it recovers. Yet, it has missed a number of updates which are carried out after its crash. Now, it is necessary that it is brought up to date with the other replicas. This requires knowing precisely the missed operations, and in which order these operations are to be carried out
- If atomic multicasting is supported by underlying distributed system, then the update operation that was sent to all replicas just before one of them crashed is either carried out at all nonfaulty replicas or by none at all. The update is carried out if the all nonfaulty replicas have agreed that the crashed replica no longer is member of the group. The crashed replica is now forced to join the group once More after it recovers from crash.
- Update operations will be forwarded to this recovered replica only if it is registered as being a member again. In order to join the group, recovered replica's state needs up to date with the rest of the group members. As a result, atomic multicasting assures that nonfaulty processes preserve a consistent view of the database, and forces settlement when a replica recovers and rejoins the group.

Virtual Synchrony

- We assume that each distributed system has communication layer within which messages are sent and received. On each node, messages are first buffered in communication layer and then delivered to the application which runs in higher layer. **Group view** is view on the set of processes in the group which sender had when message m was multicast. Every process in group should have same view about delivery of message m sent by sender. All should agree that, message should be delivered to each member in group view and to no other member.
- Suppose message m is multicast by sender with **group view** G and while multicast is going on, new process joins the group or leaves the group. Because of this group view changes due to change in group membership. This change in group membership message c (joining or leaving group) is multicast to all the members of the group. Now we have two multicast messages in transit: m and c. In this case, guarantee is needed about either m is delivered to all processes in group view G before each one of them delivered message c, or m is not delivered at all.
- When group membership change is the result of crashing of the sender of m, then delivery of m is permitted to fail. In this case, either all the members of the G should know abort of new member or none. This guarantees that, a message multicast to group view is delivered to each nonfaulty process in G. If sender crashes during multicast then message may be delivered to all remaining processes or ignored by each of them. Reliable multicast with this property is said to be **virtually synchronous**.
- Consider processes P_1, P_2, P_3, and P_4 in group view G. Group view G = (P_1, P_2, P_3, P_4). After some messages have been multicast, P_3 crashes. So group view G = (P_1, P_2, P_4). Before crash, P_3 succeeded in multicasting messages to P_2 and P_4 but not to P_1. Virtual synchrony guarantees that, its messages will not be delivered at all. This indicates that, message had never been sent before crash of P_3. Communication then proceeds between remaining members after removing P_3 from group. After P_3 recovers, it can join the group provided its state has been brought up to date.

Message Ordering

- The multicasts are classified with following four orderings.
  1. Unordered multicasts
  2. FIFO-ordered multicasts
  3. Causally-ordered multicasts
  4. Totally-ordered multicasts
- An **unordered multicast** which is reliable is a virtually synchronous multicast. In this multicast, guarantees are not assured regarding the order in which received messages are delivered by different processes. Consider the example in which reliable multicasting is offered by a library with **send** and a **receive** primitive. The receive operation blocks the calling process until a message is delivered to it.
- Consider group with three communicating processes P_1, P_2, and P_3. Following are the ordering of events at each process.

Process P_1	Process P_2	Process P_3
sends m_1	receives m_1	receives m_2
sends m_2	receives m_2	receives m_1

- Process P_1 multicasts messages m_1 and m_2 to group members. Assume group view does not change during multicast. In this situation suppose communication layer at P_1 receives first m_1 and then m_2. In contrast, communication layer at P_2 suppose receives first m_2 and then m_3. As there is no ordering constraint, messages may be delivered in the order they were received.
- In the second case of reliable **FIFO-ordered multicasts**, the incoming messages from same process are delivered by the communication layer in the same order as they have been sent. Consider communication between following four processes. In FIFO ordering, as shown below, message m_1 will be always delivered before m_2. In the same way, m_3 will be always delivered before m_4. This rule is followed by all processes in the group. If communication layer at particular process receives m_2 first and then m_1, it should not deliver m_2 till it has received and delivered m_1. On the other hand, messages received from different processes are delivered by communication layer in the order they have received.

Process P_1	Process P_2	Process P_3	Process P_4
sends m_1	receives m_1	receives m_1	sends m_3
sends m_2	receives m_2	receives m_3	sends m_4
	receives m_3	receives m_2	
	receives m_4	receives m_4	

- In **reliable causally ordered multicast**, potentially causally related messages are delivered by communication layer by considering causality between messages. If message m_1 causally precedes message m_2 then at receiver side, communication layer will deliver m_1 first and then m_2. These messages can be from same process or from different processes.

- **Total-order multicast** imposes additional constraint on order of delivery of messages. It says that, message delivery may be unordered, FIFO or causally ordered but should be delivered to all the processes in group in same order. Virtually synchronous reliable multicasting which offers totally ordered delivery messages called as **atomic multicasting.**

5.9 Recovery

5.9.1 Introduction

- It is necessary to recover from the failure of processes. Process should always recover to the correct state. In error recovery, it is necessary to replace erroneous state to error-free state. In **backward recovery,** system is brought from current erroneous state to previous state. For this purpose, a system state is recorded and restored after some interval. When current state of the system is recorded, a **checkpoint** is said to be made.
- In **forward recovery,** an attempt is made to bring system from current erroneous state to new current state from which it can carry on its execution. Forward recovery is possible if type current error occulted is known.
- In distributed system, backward recovery techniques are widely applied as mechanism to recover from failures. It is generally applicable to all systems and processes and can be integrated in middleware as general-purpose service. The disadvantage of this technique is that, it degrades performance as it is costly to restore previous state.
- As backward recovery is general technique applicable to all systems and applications, guarantee cannot be given about occurrence of failure after recovery from the same one. Hence, applications support is needed for recovery which cannot give full-fledged failure transparency. Moreover, rolling back to state such as money is already transferred to other account is practically impossible.
- As **checkpointing** is costlier due to restoring to previous state, many distributed system implements it with **message logging**. In this case, after checkpoint has been taken, process maintains logs of messages prior to sending them off. This is called **sender-based logging.** In other scheme, **receiver-based logging** can be carried out where receiving process first log an incoming message and then deliver it to the application it is currently executing.
- Checkpointing restores processes to previous state. From this state, it may behave differently than before failure had occurred. In this case, messages can be delivered in different order leading to unexpected reactions by receivers. Message logging now helps to deliver messages in the expected order as actual replaying of the events since last taken checkpoint.

5.9.2 Stable Storage

- The information needed to recover to the previous state needed to be stored safely so that it survives process crashes, site failures and also storage media failures. In distributed system, stable storage is important for recovery.
- The storage is of three types: RAM which is volatile, disk which survives CPU failure but can be lost in case of failure of disk heads. Finally, stable storage which is designed to survive from all types of failures except major calamities.
- **Stable storage is implemented with pair of ordinary disks. Each block on second drive is an exact copy of the corresponding block on first drive. Update of block first takes place in first drive and once updated, it is verified. Then same block on second drive is done.**

- Suppose first drive is updated and then system crashes. Suppose, this crash is occurred before second drive is updated. In this situation, recovery is carried out by comparing both drives block for block. When two corresponding blocks differ then block on first drive is considered as correct one and it is then copied to first drive to another drive. Now both the drive will be identical.
- Due dust particles and general wear and tear, a valid block may give checksum error. In this case, block is recovered by using the corresponding block on other (second) disk. Stable storage is most beneficial for applications that need a high degree of fault tolerance.

5.9.3 Checkpointing

- In backward error recovery, it is required to save system state regularly on stable storage. It also needed to record consistent global state called as distributed snapshot. In distributed snapshot, if one process has recorded the receipt of message then there should be other process who has recorded the sending of the message.
- In backward error recovery schemes, each process saves its state on regular basis to its local stable storage. A consistent global state from these local states is build to recover from process or system failure. A recovery line signifies the most recent consistent collection of checkpoints. The most recent distributed snapshot is recovery line.

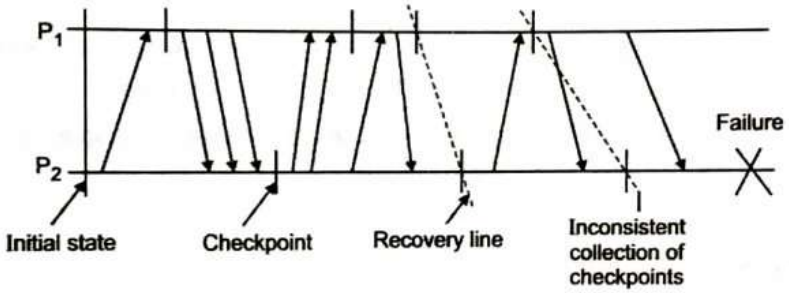


Fig. 5.9.1 : A Recovery Line

Independent Checkpointing

- If each process records its local state from time to time in an uncoordinated way then it is not easy to find recovery line. In order to find out this recovery line, each process is rolled back to its most recently recorded state. If these local states together do not form distributed snapshot then further rolling back is necessary to carry out. This process of cascaded roll back may lead to **domino effect.**

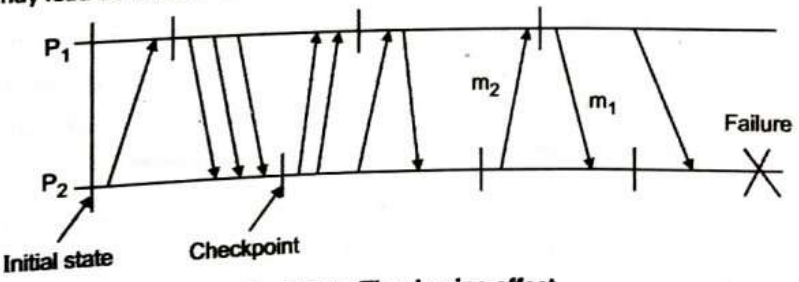


Fig. 5.9.2 : The domino effect

- After crash of process P_2, it is required to restore its state to the most recently recorded checkpoint. As a result, process P_1 will also needs to be rolled back.

- The restored state by P_2 shows receipt of m_1 but sender of the same message is still not recorded in snapshot. Again, it is require P_2 to roll back to previous state. However, the next state to which P_2 is rolled back has not recorded sending of message m_2 although; P_1 has recorded receipt of it. As a result, P_1 again needs to be rolled back.
- This process of rolling back without coordinating with other process is called independent checkpointing The other solution is to globally coordinate checkpointing which requires global synchronization. This may introduce performance problems. Another drawback of independent checkpointing is that, each local storage needs to be cleaned up time to time.

Coordinated Checkpointing

- In coordinated checkpointing, all processes synchronize to write their state to local stable storage in cooperative manner. As a result of which, the saved state is automatically globally consistent. In this way, domino effect is avoided here.
- A two-phase blocking protocol is used to coordinate checkpointing. A coordinator first multicast ***CHECKPOINT_REQUEST*** to all the processes. Receiving process of this message then takes local checkpoint, queues any successive message handed to it by the application it is running, and acknowledges to the coordinator that it is has taken a checkpoint.
- When coordinator receives acknowledgement from all the processes, it multicast ***CHECKPOINT_DONE*** message to permit blocked processes to carry on their work. As incoming messages are not registered as part of checkpoint, this approach guarantees globally consistent state. All the outgoing messages are queued locally till receipt of ***CHECKPOINT_DONE*** message.
- This approach can be improved if checkpoint request is multicast to those processes only that depend on the recovery of the coordinator. A process is dependent on the coordinator if it has received a message that is directly or indirectly causally related to a message that the coordinator had sent since the last checkpoint. This leads to the concept of an incremental snapshot.
- In an incremental snapshot, the coordinator multicasts a checkpoint request only to those processes it had sent a message to since it last took a checkpoint. When a process *P* receives such a request, it forwards the request to all those processes to which *P* itself had sent a message since the last checkpoint, and so on.

5.9.4 Message Logging

- Message logging enables recovery by reducing number of checkpoints. Checkpointing is costly in terms of writing state on stable storage. In message logging, transmission of messages is replayed to achieve globally consistent state. There is no need to write state on stable storage. In this approach, a checkpointed state is taken as starting point and all messages that have been sent since are simply retransmitted and handled accordingly.
- In order to recover from process failure, replaying of messages is carried out in message logging scheme to restore to globally consistent state. For this purpose, it is important to know exactly when messages are to be logged.
- The existing message-logging approaches can be easily distinguished on the basis of how they handle orphan processes. An orphan process is a process that survives the crash of another process, but whose state is inconsistent with the crashed process after its recovery.

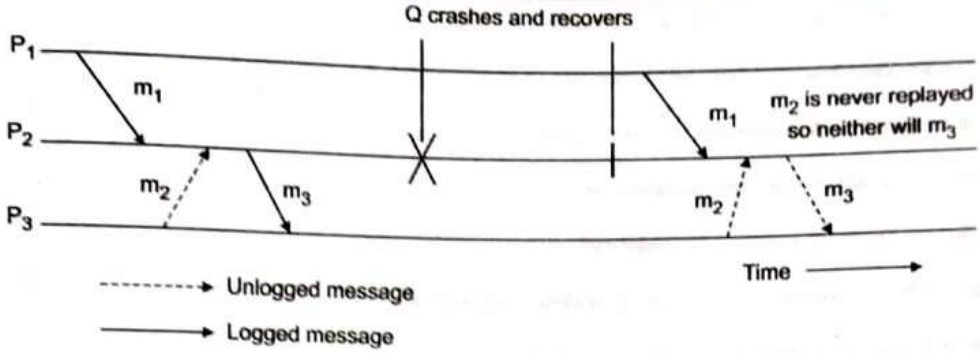


Fig. 5.9.3 : Incorrect replay of messages after recovery, leading to an orphan process

- Consider three processes P_1, P_2 and P_3 as shown in Fig. 5.9.3. Process P_2 receives messages m_1 and m_3 from P_1 and P_3 respectively. Process P_2 then sends messages m_3 to P_3. Message m_1 was logged but m_2 was not logged. After crash process P_2 recovers again. In this case, for recovery of P_2 only logged message m_1 will be replayed. As m_2 was no logged, its transmission will not be replayed and hence, transmission of m_3 may not take place.
- However, the state after the recovery of P_2 is inconsistent with that before its recovery. Especially, P_3 keeps a message m_3 which was sent before the crash, but whose receipt and delivery do not happen when replaying what had taken place before the crash. Such inconsistencies should clearly be avoided

5.9.5 Recovery-Oriented Computing

- The other way of carrying out recovery is basically to start over again. It may be much cheaper to optimize for recovery, then it is aiming for systems that are free from failures for a long time. This approach is called as recovery-oriented computing. One approach is simply reboot the system.
- For clever reboot only a part of the system, it is essential to localize the fault correctly. Just then, rebooting simply involves deleting all instances of the identified components together with the threads operating on them, and (often) to just restart the associated requests. Practically, rebooting as a recovery technique requires few or no dependency between system components.
- In other approach of recovery-oriented computing, checkpointing and recovery techniques are applied, but execution is carried out in a changed environment. The basic idea is that, if programs are allocated more buffer space, changing order of message delivery etc then many failures can be avoided.

Review Questions

Q. 1 What is replication? Write the advantages of replication.

Q. 2 Explain replication as scaling technique.

Q. 3 What is continuous consistency? Explain.

Q. 4 Explain sequential consistency model with example.

Q. 5 Explain causal consistency model with example.

Q. 6 Explain FIFO consistency model with example.

Q. 7 Explain release consistency model with example.

Q. 8 Explain entry consistency model with example.

Q. 9 What is basic difference between consistency and coherence?

Q. 10 What do you mean by eventual consistency?

Q. 11 What do you mean by eventual consistency?

Q. 12 Explain ant two client-centric consistency models with example.

Q. 13 Explain ant two data-centric consistency models with example.

Q. 14 Explain read your writes and write follows reads client centric consistency.

Q. 15 Explain permanent replicas, server-initiated replicas and client-initiated replicas.

Q. 16 Explain different techniques of propagation of updates.

Q. 17 Explain different failure models.

Q. 18 Explain Byzantine Agreement Problem with example.

Q. 19 Explain different failures that can occur in RPC and their solutions.

Q. 20 Explain different approaches for scalability in reliable multicasting.

Q. 21 What is mean by atomic multicast? Explain.

Q. 22 What is mean by atomic multicast? Explain.

Q. 23 What is virtually synchronous reliable multicast? Explain.

Q. 24 Explain different types of ordering of the messages in group communication.

Q. 25 What is stable storage? How it plays role in recovery of distributed system?

Q. 26 What is checkpointing?

Q. 27 Explain independent and coordinated checkpointing approaches.

Q. 28 What is message logging? What are the advantages of message logging?

Q. 29 Explain recovery-oriented computing.

□□□

CHAPTER 6

Module 6

Distributed File Systems and Name Service

Syllabus

Introduction and features of DFS, File models, File Accessing models, File-Caching Schemes, File Replication, Case Study: Distributed File Systems (DFS), Network File System (NFS), Andrew File System (AFS), Introduction to Name services and Domain Name System, Directory Services, Case Study: The Global Name Service, The X.500 Directory Service , Designing Distributed Systems: Google Case Study.

6.1 Introduction

- **File system** describes how files are structured, named, accessed, used, protected and implemented. Files are used for permanent use of information on secondary storage. It also provides sharing of information.
- In distributed system, files are available on several computers. Computers in distributed system can share these physically dispersed files by using distributed file system. Service offers particular function to client and it is a software entity running on some machines. Server runs service software on single machine. Client process invokes the service through some set of defined operations called as client interface.
- File system offers file services to clients. The set of primitive file operations are create a file, delete a file, read from a file, and write to a file. Client interface is formed with set of these operations. File server controls local secondary-storage devices such as disks on which files are stored. These files are accessed from these devices as per request of client.
- There can be different implementation for distributed file system (DFS). Server may run on dedicated machines. In other implementation both client and server may run on same machine. DFS can be part of distributed operating system or it can be a software layer managing communication between file system and network operating system. DFS should come into view to its client as conventional centralized file system. The servers and storage devices which are on different machines in network should be invisible to clients. DFS should fulfil the request of client by arranging the required files or data.
- As data transfer is involved in operation of DFS, its performance is measured with amount of time required to service the client request. Storage space managed by a DFS includes different and remotely located small storage spaces.
- DFS supports the following :
  - **Remote Information sharing :** Any node in the system can transparently access file irrespective of its location.
  - **User mobility :** DFS allows the user to work on different nodes at different times without physically relocating the secondary storage devices.
  - **Availability :** DFS keeps multiple copies of file on different nodes. Failure of any node or copy does not affect the operation.

- **Diskless workstations :** DFS provides transparent file accessing capability; hence, economical diskless workstations can be used.

- DFS provides following three types of services.
  - **Storage service :** DFS deals with storage and management of space on secondary storage device that is used to store files in file system. Files can be stored in blocks. It offers logical view of storage system by allowing operations for storing and accessing the data in them.
  - **True File service :** True file service supports operations on individual file. These are creating and deleting files, modifying and accessing data from files. Typical design issues to support these operations include file accessing techniques, file sharing semantics, file caching and replication mechanism, data consistency, multiple copy update protocol and access control mechanism.
  - **Name service :** Name service offers mapping between file names to file IDs. It is difficult for human to remember file IDs. Most file system uses directories to carry out mapping called as directory service. This service supports creation and deletion of directories, adding new file to directory, deleting the existing file from directory, changing name of the file etc.

6.2 Desirable Features of a Good Distributed File System

Following are the features of good DFS :

1. Transparency

- **Structure transparency:** DFS uses multiple file servers. Each file server is user or kernel process to control secondary storage devices of that node on which it runs. Client should be unaware about location and number of file servers and storages devices. DFS should treat the client just like single conventional file system offered by centralized time sharing OS.
- **Access Transparency:** Accessing the local and remote file should be carried out in similar manner. DFS should automatically locate accessed file and support to transporting the data to client.
- **Naming Transparency:** name of file should remain same when it is moved from one node to other. Its name should be location independent.
- **Replication Transparency:** Existence of multiple replicated copies and their location should remain hidden from clients.

2. User mobility

DFS should permit the user to work on different nodes at different times. Performance should not affect if user works on node other than his node. User's home directory should be automatically made available when user logs in on new node.

3. Performance

DFS must give performance same as centralized file system. User should not feel the need to place file explicitly to improve performance.

4. Simplicity and ease of use

User interface to file system should be simple and small number of commands should be supported. It should support for all types of applications. The semantics of DFS should be same as single conventional file system for centralized time sharing system.

5. **Scalability**

The DFS should also supports for growth of nodes and users in network. Such growth should not lead to service loss or performance degradation of the system.

6. **High Availability**

DFS should continue to function if partial failure occurs in one or more components. This failure can be communication link failure, node failure, and secondary storage failure. Degradation in performance due to failure must be in proportional to that failure.

7. **High Reliability**

DFS should support for reliability. It should automatically create backup copies of the critical files that can be loss in the failure. Users should ensure that there will be no or minimum loss of their data.

8. **Data Integrity**

Simultaneous accesses to shared file by many users should guarantee the integrity of data stored in it. These access requests to file from many users should be properly synchronized by using proper concurrency control mechanism.

9. **Security**

DFS should be secured so that it can offer the privacy to user's data. It should implement the security mechanism to protect file data from unauthorized access. Secured access right allocation policy should also be supported.

10. **Heterogeneity**

In large distributed system, machine's architecture and platforms used are different. DFS should support for heterogeneity so that users can use different computer platforms for different applications. DFS should allow to integrate a new type of node or storage area in simple manner.

6.3 File Models

Following are two file models based on structure and modifiability :

6.3.1 Unstructured and Structured Files

- **Unstructured Files :** File is unstructured sequence of data. Structure of the file appears to file server as un-interpreted sequence of bytes. The interpretation of structure and meaning of data in file is up to the program and OS is not interested in the same. This file model is used by MS-DOS and UNIX. Most of the operating system used this model as sharing of files by different applications is easier. Different application can view content of the files in different way.
- **Structured Files :** In this model, file is presented to file server as order sequence of records. Record is smallest unit of data that can be accessed. Different files in the same file system may have different size of records. To access the record in structure files with non-indexed records, its position within needs to be specified. For example seventh record from beginning of file. In structure files with indexed records, a record is accessed by using key value.
- File is identified by its name which is string of the characters. Every file has a name and its data. All operating systems associate other information with each file, for example, the date and time of file creation and the size of the file. Such extra information associated by operating system is called as attributes.

- The lists of attributes are not same for all the systems and vary from one operating system to another. No single existing system supports all of these attributes, but each one is present in some system.

6.3.2 Mutable and Immutable Files

- **Mutable Files :** Most of the OS uses this model. In this file, each update operation on file update overwrites its old content and new content is produced. Hence, file is represented as single stored sequence that is changed by each update operation.
- **Immutable Files :** In this file model, each update operation creates new version of the file. Changes are made in new version and old version is retained. Hence, more storage space is required.

6.4 File-Accessing Models

6.4.1 Accessing Remote Files

DFS may use one of the following models to service request of the client to access the file.

- **Remote Service Model :** In this model suppose client forward the request to server to access remote file. Naming scheme locates the server and actual data transfer between client and server is achieved through remote-service mechanism. In this mechanism, request for accesses is forwarded to server, which then performs accesses and returns the result to user or client. This is similar to disk accesses in conventional file system. Hence, data packing and communication overhead is significant in this model.
- **Data-Caching Model :** Caching can be used to improve performance of remote-service mechanism. In conventional file system, caching is used to reduce disk I/O. The main goal behind caching in remote-service mechanism is to reduce network traffic and disk I/O. If data is not available locally then it is copied from server machine to client machine and cached there. This cached data is then used by client at client side to process its request. This model offers better performance and scalability.
- All the future repeated accesses to this recently cached data can be carried out locally. This will reduce additional network traffic. Least Recently Used (LRU) algorithm can be used for replacing the cached data. Master copy of file is available on server and its part is scattered on many client machines. If copy of the file at client side modifies then its master copy at server should be updated accordingly called as cache-consistency problem.
- DFS caching is network virtual memory which works similar to demand-paged virtual memory having remote server as backing store. In DFS, data cached at client side can be disk blocks or it can be entire file. Actually more data are cached by client than actually needed so that most of the operations are carried out locally.

6.4.2 Unit of Data Transfer

The unit of data transfer is fraction of data that is transferred between client and server due to single read or write operation. In data-caching model of accessing remote files, following four models are used to transfer the data.

File-level Transfer Model

- In this model, whole file is transferred in either direction across the network. Hence, the network protocol overhead is required only once. Its scalability is better as it requires smaller number of accesses to serve and hence, reduces server load and network traffic.
- Disk access routines on server always better optimized as it is known that accesses are for file instead of random blocks.
- One time transfer is required to transfer the file to the form compatible to client file system. The drawback of this form of transfer is that it requires sufficient space at client as entire file is cached. Amoeba, CFS and Andrew File System uses this transfer model.

Block-level Transfer Model

- Transfer of file data between client and server takes place in units of file blocks. Usually, File is stored in continuous blocks of fixed size on secondary storage. In this model, no need to transfer entire file to client machine and hence, it is better alternative approach to diskless workstations.
- But, if client needs access to entire file then all blocks needs to be transferred and hence, it generates more network traffic and has poor performance. NFS, LOCUS and Sprite use this transfer model.

Byte-level Transfer Model

In this model, transfer of file data between client and server takes place in units of bytes. It offers flexibility as any range of data bytes within file can be requested and hence, it is difficult to manage cached data due to its variable length. Cambridge file server uses this model.

Record-level Transfer Model

This model is applicable to structured files only. Hence, records are transferred between client and server. Research storage system (RSS) uses this model.

6.5 File-Caching Schemes

- These schemes are basically for retaining the cached data of recently accessed files in main memory in order to reduce repeated disk accesses for the same data. It gives good performance due to reductions in disk transfers.
- Hence, it is good approach to improve performance of the file system. It also helps to achieve scalability and reliability as remote data is possible to cache on client node. Therefore, file caching is used by most of DFS.
- File caching scheme for DFS should address the decisions like cache location, propagation of modifications and validation of cache.

6.5.1 Cache Location

- Suppose the original location of the file is server's disk. Following three possible locations can be there to cache the data.

- **Server's main memory :** This is no caching scheme before client can access the file, the file is first transferred from server's disk to the server's main memory and then to the clients main memory across the network. In this, one disk access and one network access is required. It results in good performance as it eliminates disk access. It is easy to implement. As file resides on the same server machine, it is easy to maintain consistency between original file and cached data. The drawback is that, client has to carry out network access for each file access. Hence network cost is involved. It is not scalable and reliable approach.
- **Client disk :** It involves disk access cost at client machine on cache hit. It eliminates network access. In case of crash, data remains in client disk and hence, no need to access again from server for recovery. There is no loss of data as disk is permanent storage. Hence, it offers reliability. Disk also has large storage capacity compared to main memory, resulting in higher hit ratio. Most of the DFS uses file level transfer for which caching in disk is better solution as disk has large storage space for file. The drawback is that, this policy does not work for diskless workstations.
- **Client's main memory :** It works for diskless workstations and avoids network access cost and disk access cost. It contributes scalability and reliability as access request is served locally on cache hit. It is not preferable compared to client disk cache if increased reliability and large cache size is required.
- Cache location can either be main memory or disk. If cache is kept in main memory then modifications done on cached data will lost due crash. If caches are kept in disk then they are reliable. No need to fetch the data during recovery as data resides on disk. Following are the advantages of main memory caches.
  - It permits for diskless workstations
  - Data access takes less time from main memory compared to access from disk.
  - Performance speed up is achieved with larger and inexpensive memory which technology demands today.
  - To speedup I/O server caches are kept in main memory. If both server caches and user caches are kept in main memory then single caching mechanism can be used for both.
- Many remote-access implementations take the hybrid approach considering both, caching and remote service. In NFS, for example, the implementation is based on remote service but is improved with client- and server-side memory caching for performance.

6.5.2 Modification Propagation

- If caching of the data is at client side then file's data may simultaneously be cached on multiple client nodes. If all these cached data copies are exactly same then caches are consistent. If one client changes file data and corresponding data on other client nodes are not changed then caches becomes inconsistent.
- To maintain consistency between all the copies of file data, several approaches are available. These approaches depend on the schemes used for following cached design issues for DFSs.
  - When to propagate modifications made to cached data to file server.
  - How to verify validity of cached data.
- System's performance and reliability depends on the policy used to write back updated data to the master copy of file which is available on server.
- Following policies are used :

Write-through policy

- This policy sends modified cache entry immediately to the server for updating the master copy of the file. This policy is more reliable as little information is lost when client system crashes. The write performance is poor as each write access has to wait until the information is sent to the server.
- The advantages of data caching are only for read accesses as remote service method is used for all write accesses. It is suitable in cases where read to write access ratio is quite large.

Delayed-write policy or write-back caching

- In this policy there is delay in writing the modifications to the master copy on server. Modifications are written first in cache and then later time is done on master copy. First advantage of this policy is that write accesses complete in less time as writes are made to cache.
- Second, the data which may be overwritten prior to writing back on master copy, so last update needs to be written. The limitation of this policy is that, if client crashes then unwritten data are lost and hence less reliable.
- There are variations of delayed write policy. One choice is to flush a block as soon as it is set to be ejected from the client's cache. This alternative can lead to good performance, but some blocks can exist in the client's cache a long time before they are written back to the server.
- A negotiation between this choice and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan. So far one more variation on delayed write is to write data back to the server when the file is closed. This **write-on-close** policy is used in Andrew File System (AFS).

6.5.3 Cache Validation Schemes

- Client machine always use cached data for accesses which is consistent with master copy at server. If client determines that its cached copy is out of date, then it should cache the up-to-date copy of data.
- Following two approaches are used :

Client Initiated Approach

- The client starts a validity check in which it contacts the server. In this check, client checks consistency of its cached data with the data in master copy. The resulting consistency semantics is decided by frequency of validity checking.
- Validity checking can be carried out prior to every access or on first access to a file or at regular intervals. The validity check is carried out by comparing time of last modification of the cached version of data with server's master copy version. If two are same the cached data is considered as up to date. Otherwise recent version is accessed from the server.

Server Initiated approach

- The server maintains the records of the files or portion of files that each client catches. Server must take immediate action whenever it detects a potential inconsistency. If two different clients in conflicting modes cache a file then there is possibility of inconsistency.

- A notification should be sent to server when a file is opened, and the intended read or write mode must be specified for every open. Whenever server detects that file has been opened simultaneously in conflicting modes, it can take action by disabling caching for that particular file. When catching is disabled then a remote-service mode of operation becomes active.
- This policy has some problems also. It violets traditional client server model. Hence, it makes code for client and server programs irregular and complex. It also requires stateful file server which has limitations compared to stateless file servers in case of failures. A validation is carried out with check on open policy which is client initiated approach; it should be used along with server initiated approach.

Comparison of Caching and Remote Service

Sr. No.	Caching	Remote Service
1.	It allows to serve remote accesses locally so that they can be faster as local accesses	Remote access is handled across the network so slower compared to caching.
2.	It reduces the network traffic, server load and improves scalability as server is contacted occasionally.	It increases network traffic, server load and degrades performance.
3.	In case of caching, for transmission of big chunks of data, overall network overload required is at lower side compared to remote service.	In case of remote service, transmission of series of responses to specific requests involve higher network overhead as compared to caching.
4.	Caching is better in case of infrequent writes but if writes are frequent then mechanism used to overcome consistency problem incur large overhead in terms of performance, network traffic, and server load.	In remote service, there is always communication between client and server to have a master copy consistent with client's cached copy.
5.	Caching is better option for machines with disk or large main memory.	If machines are diskless and with small main memory then remote service should be carried out.
6.	In case of caching, the lower-level inter-machine interface is different from the upper-level user interface.	The remote service concept is just an extension of the local file-system interface across the network.

6.6 File Replication

- Replicating the file on many machines improves availability. If nearest replica is used then service time also reduces. The replica of the same file should be kept on failure independent machines so that availability of one replica is not affected by availability of remaining other replica. Replication of files should be hidden from users.
- It is the responsibility of naming scheme to a replicated file name to a particular replica. Higher levels should remain invisible from existence of replicas.

- At lower-level different lower-level names are used for different replicas. Replication control such as determination of the degree of replication and placement of replicas should be provided to higher levels. As one replica updates then from user's point of view, the changes should be reflected in other copies as well.

6.6.1 Replication and Caching

- In replication, replica is created at server, whereas cached copy is associated with clients.
- A replica is more persistent as compared to cached copy. Replica is widely known, secure, accurate, available and complete.
- Cached copy existence depends on locality in access patterns. Existence of replica depends on availability and performance.
- Cache copy is dependent on replica. It needs to be periodically verified with replica then it becomes useful.

6.6.2 Advantages of Replication

Following are the advantages of replication :

1. **Increased Availability :** Replication offers availability of the system. Although failure occurs, the system remains operational and available to users. The critical data can be replicated on several servers. If primary copy fails, still it can be accessed from other server.
2. **Increased Reliability :** As several copies of the files are available on different servers, recovery in case of failure is possible. Permanent loss of data due to catastrophic failure is also easy to recover from other replicated copy. Hence, replication offers reliability.
3. **Improved Response Time :** Replication allows data to be accessed locally or from the node whose access time is less than that of primary copy access time.
4. **Reduced Network Traffic :** If replica of file is available with file server that resides on client machine then client access request is serviced locally. Hence, it results in reduced network traffic.
5. **Improved System Throughput :** As different client's requests are serviced by different servers in parallel, it results in improved throughput.
6. **Better Scalability :** If file is replicated at multiple file servers then all the client's requests for that file would not arrive at one file server. In this way, load gets distributed and different client's requests are serviced by different servers. It improves scalability.
7. **Autonomous Operation :** All the files needed by clients for limited time period can be replicated on file server that resides on client machine. This ensures temporary autonomous operation of client node.

6.6.3 Replication Transparency

- User should not be aware about file replication. Although file is replicated, it should appear as a single logical file to its user.
- There should be same client interface for replicated and non-replicated file as well. Following are the two important issues related to replication transparency are naming of replicas and replication control.

Naming of Replicas

- As immutable objects are easily supported by kernel, single identifier can be assigned to all the replicas of immutable object. As all copies are identical and immutable, kernel can use any copy. In case of mutable objects, all copies may not be consistent at particular instance of time.
- If single identifier is assigned to all replicas of mutable object then kernel cannot decide which replica is most up-to-date. Therefore consistency control and management for mutable objects should be carried out outside the kernel.
- Naming system should map a user supplied identifier to the appropriate replica of mutable object. In case if all the replicas are consistent then mapping must provide location of the replicas and their distance from client node.

Replication Control

- Replication control is transparent from user and handled automatically. Replication can be carried out system automatically or can be carried out manually.
- In **explicit replication**, users control the process of replication. Created process specifies server on which file should be placed. If needed then additional copies are created as per request from users. Users also have flexibility to delete one or more replica. In **implicit replication**, entire process of replication is controlled by system automatically. Users remain unaware of this process. Server to place the file is selected by system automatically. System also creates and deletes replicas as per replication policy.

6.6.4 Multicopy Update Problem

As replicas of file exist on multiple machines, it is necessary to keep all the copies consistent. If update takes place on one copy, it must be propagated to all other copies. Following approaches are used :

Read-Only Replication

In this approach, only immutable files are replicated as they are used only in read only mode. These file gets updated after longer period of time.

Read-Any-Write-All Protocol

This protocol allows replication of mutable files. In this approach, read operation is performed on any copy of the file but write operation is performed by writing to all copies of file. Before performing update to any copy, all copies are locked, then they are updated, and finally locks are released to complete the write.

Available-Copies Protocol

- In read-any-write-all protocol, if server with replicated copy is down at the time of write operation then write cannot be performed. Available-copies protocol allows this operation. In this approach, read operation is performed by reading any available copy of the file but write operation is performed by writing to all available copies of file.
- The assumption in this protocol is that, down server when recovers, it brings its state up-to-date from other server's copies. This protocol provides high availability but does not prevent inconsistencies in failure of communication links.

Primary-Copy Protocol

- In this protocol for each replicated file, one copy is kept as primary and all other copies are considered as secondary copies. The write operation is carried out only on primary but read operation can be carried out on any copy including primary.
- Server with secondary copy either receives notification of updates from server with primary copy or updates are requested by server with secondary copy itself. Primary copy immediately informs updates to secondary copies if any occurs to it.

Quorum-Based protocol

- This protocol is applicable in network partition problem where replicated file are partitioned in two more active groups. All above protocols discussed has some restrictions.
- Following are the definitions of read and write quorum.
  - (i) **Read quorum :** To read a file, if minimum r copies of replicated file F have to be consulted out of n replicated copies of F then set of r copies is called as read quorum.
  - (ii) **Write quorum :** To carry out write operation on the file, if minimum w copies of replicated file F have to be written out of n replicated copies of F then set of w copies is called as read quorum.
- Restriction in this protocol is that sum of r and w should be greater than n (r+w>n). It ensures that, between any pair of read-quorum and write-quorum, there is at least one copy common which is up-to-date. This protocol needs to identify current updated copy to in order to update other copies of the file. This problem is resolved with assigning the version number to copy when it gets updated. Highest version number copy in quorum is current updated copy. New version number to be assigned is one more than the current version number.
- In **read** operation, only highest version number copy is selected to read from read quorum. For **write** operation, only highest version number copy is selected from write quorum to carry out write operation. Before performing write, the version number is incremented by one. Once update is carried out, the new update and new version number is written to all the replicas.
- Consider n = 8, r = 4, and w = 5. In this example, r+w>n condition is satisfied. If write operation is carried out on write quorum {3, 4, 5, 6, 8} then these copies are updated versions with new version number. If read operation is performed on read quorum {1, 2, 7, 3} then copy 3 is common copy as read operation must contain at least one copy of previous write-quorum. This exactly ensures copy 3 as having largest version number from read-quorum. Hence, read is carried out with copy 3.
- Following are few special alternatives suggested for this protocol :
  - **Read-Any-Write-All Protocol :** Suitable to use when read to write ratio is high in case of r = 1 and w = n.
  - **Read-All-Write-Any Protocol :** Suitable to use when write to read ratio is high in case of r = n and w = 1.
  - **Majority-Consensus Protocol :** Suitable to use when read to write operations is nearly 1. In this protocol, read and write quorum is of same size or of nearly equal size. If n=12 then r = 6 and w = 7, or If n=11then r = 6 and w = 7.
  - **Consensus with Weighted Voting :** In this protocol, different replicas are assigned with different votes considering reliability and performance.

- If replica X is accessed more frequently then more votes are assigned to it. Size of quorum depends on replicas selected for quorum. In this protocol, to ensure non-null intersection of read and write quorums, condition r + w > v where v is total number of votes.

6.7 Case Study : Distributed File Systems (DFS)

- DFS forms basis for many distributed application and sharing of data is fundamental to it. DFS supports to share data by multiple processes over long period of time while offering security and reliability.
- SunMicro system's Network File System (NFS) and Andrew File system (AFS) are the examples of distributed file system.

6.7.1 Network File System (NFS)

- NFS is developed by Sun Microsystems and it is used on Linux to join file systems of different computers into one logical whole. Version 3 of NFS was introduced in 1994. NSFv4 was introduced in 2000 and offers a number of improvements over the previous NFS architecture.
- NFS allows a number of clients and servers to share a common file system. These clients and servers can be on the same LAN or in wide area network if server is located geographically at long distance. NFS permits every machine to be both a client and a server simultaneously.

NFS Architecture

- NFS uses remote file service model. Remote server manages all the accesses of clients. NFS offers interface to clients similar to conventional local file system to access the files transparently. All the file operations are implemented by server but their interfaces are offered to the clients. It is **remote service model**. NFS also supports **upload/download model** in which client downloads the file for operations and then uploads it to the server so that updated file can be used by other clients.
- UNIX based version of NFS is predominant version. Client uses system calls of its local operating system (UNIX). The local UNIX file system interface is replaced by interface to virtual file system (VFS). The operations carried out to VFS interface is either passed out to local file system or to **NFS client** component. NFS client then performs remote procedure call (RPC) to access files at remote server.This means NFS client implements file operations as RPC to remote server. VFS hides the differences between different file systems.
- NFS server on server side handles incoming requests. RPC server stub unmarshals these requests and NFS server convert them to regular VFS file operations that are afterward passed to the VFS layer. VFS implements local file system where actual files reside. In this way, NFS is independent of local file systems, provided, if local file system compliant with file system model of NFS.

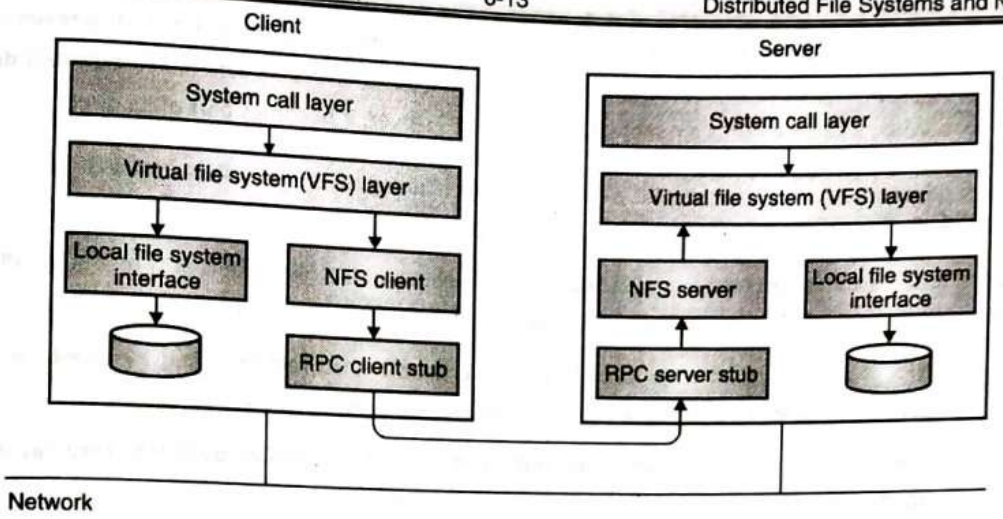


Fig. 6.7.1 : Basic NFS Architecture for UNIX Systems

File System Model, Communication and Naming

- In NFS files are considered as uninterpreted sequence of bytes. Files are organized in naming graph in which nodes represent files and directories. NFS supports hard link or symbolic links just like UNIX file systems.
- Files are accessed by means of UNIX like file handle. At first, client searches file name by using naming service to obtain the file handle. Each file has a several attributes whose values can be found and changed. These attributes includes length and type of file, It's file system identifier, and last time file was modified.
- For reading the data from file, read operation is used. Client specifies offset and number of bytes to read. Writing data to file is carried out with write operation. The position and number of bytes to write is specified by client.
- For communication, NFS protocol is placed on top of RPC layer. This is due to independence of NFS on OS, network architecture and transport protocols. Open network computing remote procedure call **(ONC RPC)** is used for the communication. NFS supports to group several RPC requests in one request in order to reduce number of messages to be exchanged.
- This is called as **compound procedures** which do not have any transactional semantics. All the operations in compound procedure is executed in order of their requests. Conflicts cannot be avoided in case if same operations are invoked by other clients simultaneously.
- In NFS, stateless server was implemented initially which cannot supports to lock a file for operations. Therefore, a separate lock manager is used to handle same situation. In later version, stateful approach is implemented to support working across wide area network so that clients can use caches effectively.
- NFS naming model offers transparent access for remote file system at server to its requesting clients. NFS allows clients to mount part of file system at server. The exported directory by server can be maintained in client's local space. The drawback of this approach in DFS is that it does not allow sharing files.
- Remote clients access the NFS server's exported directories. Every NFS server exports one or more of its directories. These directories then accessed by remote clients. Particular directories along with its subdirectories, so in fact entire directory trees are usually exported as a unit.

- The directories list that server exports are kept in a file, often in */etc/exports* in case of Linux. As a result of this, these directories can be exported automatically whenever the server is booted. Clients can access exported directories by mounting them. When a client mounts a remote directory at NFS server, it becomes part of its directory hierarchy and client can access this mounted directory.

File Handle and Automounting

- It is reference to a file within file system and created by server hosting the file system. It is unique to all file systems exported by server. It is created at the time of creation of file.
- In version 4 of NFS, it can be variable up to 128 bytes. File handle is stored and used by client for most of the operations, and hence, avoids look up for file which improves performance.
- After deleting file, server cannot reuse the same file handle as it can be locally stored by client. This may lead to wrong file access by using the same file handle by client.
- Iterative look up with not permitting look up operation to cross a mount point leads to the problem in getting initial file handle. To access the file in remote file system, client must provide file handle of directory where look up would take place. This also requires providing name of the file or directory that is to be resolved.
- NFS version 4 solves this problem by offering a separate operation *pootrootfh* that informs server to solve all file names relative to root file handle of the file system it manages. In NFS, on demand mounting of remote file system is handled by automounter which runs as separate process on client machine.

File Attributes

- In NFS version 4, attributes are classified as mandatory attributes that every implementation must support, set of recommended attributes which should be preferably supported and additional set of named attributes. The named attributes are encoded as an array with entry as attribute-value pair. These are not part of NFS protocol. Following are the examples of some mandatory attributes.
  - **TYPE :** Type of file (regular, directory, symbolic link).
  - **SIZE :** Length of file in bytes
  - **CHANGE :** Indicator for client to check if and/or when the file has changed.
  - **FSID :** Server-unique identifier of file's file system.
- Following are some of the general recommended file attributes :
  - **ACL :** Access control list associated with file.
  - **FILEHANDLE :** Server provide file handle of this file
  - **FILEID:** File system unique identifier for this file.
  - **FS-LOCATIONS :** Locations in network where this file system can be found.
  - **OWNER:** Owner of the file.
  - **TIME_ACCESS :** Last time of accessing the data of file.
  - **TIME_MODIFY :** Time when file data were last modified.
  - **TIME_CREATE :** Time when file was created.

Synchronization

- Distributed file system allows to share its files among multiple clients. These shared files between multiple users should remain consistent. For this purpose, synchronization is needed. In NFS file locking is carried out by separate lock manager. In version 4 of NFS file locking is integrated in NFS file access protocol. It may happen that client or server may fail while locks are still held. Recovery mechanism should handle such problem to ensure consistency.
- Four locking operations are defined in NFS version 4 for locking. Same file can be shared by multiple clients if they only read data. It is necessary to obtain write lock to access to modify part of the file data.
- Following operations are provided for locking :
  - **Lock :** Create lock for range of bytes.
  - **Lockt :** Test whether conflicting lock has been granted.
  - **Locku :** Remove a lock from range of bytes.
  - **Renew :** Renew a lease on specified lock.
- Lock is nonblocking operation and used to request a read or write lock on consecutive range of bytes in file. In case of conflicting lock, lock cannot be grated and client has to poll the server later time. Once conflicting lock has been removed, server grant the next lock to the client at top of requesting FIFO list maintained at server side. Lockt is used to check whether any conflicting lock exist. Removing lock id done by using Locku. if client does not renew lease on acquired lock, server will automatically removes it.

Replication and Consistency

- In NFS version 4, cache consistency is handled in implementation-dependent manner. Client has memory cache to hold data read from server. In extension to memory cache, disk cache also exists in client machine. Client caches file data, attributes, and directories and file handles. Client caches data obtained from server while performing several read operations. Several clients on the same machine may share the cache. If modifications are done on data then cached data is flushed back to server.
- If part of file data is cached, then whenever client opens previously closed file, it should revalidate it. Server may hand over some rights to client so that client can locally handle open, close operations. Sever is in charge of checking whether opening file by client should succeed or not.
- Clients can also cache attribute values which can be different at different clients. Modifications to attribute value should be immediately forwarded to server. Same approach is used for file handles and directories.
- NFS version 4 offers minimum support for file replication. Only replication of whole file system is possible.

Fault Tolerance and Security

- RPC mechanism in NFS does not ensure guarantee regarding reliability. It lacks in detecting the duplicate messages. In case of loss of server replay, server will process retransmitted request by client. This problem is solved by means of duplicate-request cache implemented by server. Each client request carries transaction identifier (XID) that is cached by server when request arrives at server. After processing the request, server also caches reply.

- After timer at client expires before reply comes back then client retransmits same request with same XID. Three cases occur. If server has not yet completed original request, it ignores retransmitted request. In other case, server may get retransmitted request after reply sent to client. If arrival time of retransmitted request and time at which reply sent are nearly equal then server ignores retransmitted request. If reply is really did get lost then cached reply is sent to client as reply to retransmitted request.
- In case of locks, if client is granted a lock and it crashes. In this case, server issues lease on ever lock. If not renewed by client then server removes lock freeing the resources held by lock. In case of server failure, a grace period is provided to server after it recovers. In this grace period, clients can reclaim same lock that was previously granted to them.
- As a security measure, older NFS used Diffi-Hellman key exchange to establish a session. NFS version 4 uses authentication protocol Kerberos. RPCSEC_GSS secured framework is also supported foe setting up secured channels. Authorization in NFS is analogues to secure RPC. ACL file attribute is used to support access control.

6.7.2 Andrew File System (AFS)

- UNIX programs can transparently access the remote shared files. Like NFS this transparency is offered by Andrew file system (AFS) to UNIX programs. Normal UNIX primitives are used by UNIX programs to access AFS files without carrying out any modifications or recompilation. AFS is compatible with NFS. File system in server is NFS based. Hence, File handles are use for reference of file. Remote access to file is provided via NFS.
- Scalability is major design goal of AFS. It supports to large number of active users. The whole file is cached at client node. Following are two design characteristics.
  - **Whole-file serving :** The entire content of directory and file are transferred to client machine by AFS servers.
  - **Whole-file caching :** This transferred file by server is then stored on the local disk which is permanent storage. Cache contains several recently used files. Open request are carried out locally.
- Following is the operation of AFS :
  - If file is not available on client machine. Suppose user process issues open system call for the file in shared file space. Now server is located and request is sent for copy of file.
  - The copy is then stored in local UNIX file system in client machine. The copy is then opened and its file descriptor is sent to client.
  - Processes in clients perform operation on local copy of the file. When process in client issue close system call, if the local copy is updated then its content is sent to server. Server updates the file content and timestamp on the file. The local copy is stored on client's disk for future use.

Implementation

- AFS implementation contains two software components which are UNIX processes called as Vice and Venus. Vice process runs on server machine as user-level UNIX process.
- Venus process runs on client machine as user-level UNIX process. At client side, local and shared files are available. Local files are handled as normal UNIX files. Shared files are stored on server and cached by clients in disk cache.

- UNIX kernel on each client and server is modified version of BSD UNIX. These modifications are done to interpret open, close and other system calls when they refer to files in shared name space and pass them to Venus process in client's machine. At client side, one of the file partitions is used as cache storing files cached copies of files from shared space. The management of cache is carried out by Venus process.
- It removes LRU (least recently used) files so that new files accessed from server gets space. The size of disk cache at client is large. Vice server implements the flat file service. Venus processes at client side implements hierarchic directory structure that is needed by UNIX user programs. Each file in shared file space is assigned with 96-bit file identifier. These files are identified by this identifier. It is the job of Venus process to translate path name to file identifier.

Cache Consistency in AFS

- Vice process at server side provides call back promise whenever it transfers file to Venus process running at client side. In fact, call back promise is token that guarantees that, whenever any client updates file copy, it will be notified to Venus process.
- These tokens are stored in two states along with cached file on disk cache at client machine. These are : valid and cancelled state. When server carries out a request to modify file, it notify to each client (Venus processes) to which token was issued. This notification (call back) is RPC from server to Venus process. After receiving call back by Venus process, it sets the call back promise as cancelled.
- For open operation on file, although file is available in cache, Venus checks its state. If it cancelled state then recent copy of file is fetched from server. For valid state, cached copy is then opened for further operations.
- Vice is user level process and server is dedicated to offer AFS service. UNIX kernel is modified in AFS host so that it can handle file operations by using file handles instead of conventional file UNIX file descriptors.

6.8 Introduction to Name Services and Domain Name System

- Name service is used by client processes to locate objects or to obtain addresses of resources by submitting their names. Name services often used to hold addresses and other information regarding users, machines, network domains and remote objects etc.
- Names are used to refer to resources in distributed system. These resources comprise computers, files, web pages, services and many more. URL is the example of name which is used to access web page. These names of entities are organized in name space.

6.8.1 Names, Identifiers and Addresses

- In distributed system names are used to refer to entity. Name is string of bits or characters. There are many entities exist in distributed system such as messages, web pages, mailboxes, network connections etc. It also includes resources such as files, printers, disks, computers etc.
- In order to operate on these entities, they need to be accessed. To access the entity we need access point. The name of access point is address. When mobile computer changes location, a different IP address is assigned to it. It indicates that entity may change its access time in course of time.

- Address is also a name. Name should be location independent. Name of the entity should be independent from its address.
- Identifier is also a name which is used to identify the entity. Identifier should refer to one entity and each entity should refer to by at the most on identifier. Identifier should always refer to the same entity.
- In many computer systems, addresses and identifiers are represented in the form of bit strings. Human friendly name such as URL is represented as string of characters. Many of the names are specific to some service.

6.8.2 Name Services and the Domain Name System

- Name service maintains collection of one or many naming contexts. Name service supports name resolution. Name resolution is to look up attributes from a given name. As distributed system is open, name management is separated from other services.
- Same naming scheme should be used for the resources managed by different services. Sometimes resources created in different administrative domains can be shared. Therefore common name service is required.
- Originally name services were quite simple as they were designed for single administrative domain. Considering the large distributed system with interconnection of networks, a larger name-mapping is required. Global name service is the example of naming service.
- Global name service and handle service are the two examples of name service that concentrated on scalability to large number of objects. Internet Domain Name Service (DNS) is widely used.

Name Spaces

- Names in distributed system are organized in name space. Name space can be represented as labeled diagraph having two types of nodes. Leaf node represents named entity and it has no outgoing edges. This leaf node stores information about entity such as address. It also stores state of the entity, for example, in file system it contains complete file it is representing. Directory node in name space has number of outgoing edges. Directory node maintains directory table in which outgoing edge represented as pair (edge label, node identifier).
- Naming graph contains root node. The path in naming graph contains sequence of labels. For example, path N:<label 1, label 2, ..., label n> contains N as first node in graph. Such sequence is called as path name. If first name in path name is the root of naming graph then it is called as absolute path. Otherwise it is called as relative path.
- DNS names are called as domain names. They are strings just like absolute UNIX file names. DNS name space has hierarchic structure. A domain name comprises of one or more strings called as labels. They are separated by delimiter "." (dot). There is no delimiter at beginning and end of domain name. Prefix of name is initial section of name. For example *dcs* and *dcs.qmw* are both prefixes of *dcs.qmw.ac.uk*. DNS servers do not recognize relative name. All the names referred to the global root.
- In general, alias is similar to UNIX like symbolic link which allows substituting the convenient name in place of complicated one. DNS permit aliases in which one domain name is defined to stand for another. Aliases provide the transparency. Aliases are generally used to specify machines name on which FTP server or web server runs. Alias is updated in DNS database suppose web server is moved to another machine.

- Naming domain is name space for with there is single administrative authority for assigning names within it. Domains in DNS are collection of domain names. Only some domain names identify the domain. A machine can have same name as domain. Naming data belonging to different naming domains are stored by distinct name servers managed by corresponding authorities

Combining the Name Spaces

- It is possible to embed part of one name space in other name space. Mounting file in UNIX and NFS is the example of the same. The entire UNIX file systems of two different machines can be merged by replacing each file system's root by super root. Then mount each machine's file system in this super root. In this way name spaces can be merged by creating higher level root. It raises the problem of backward compatibility.
- DCE (Distributed Computing Environment) name space permits heterogeneous name spaces to be embedded in it. DCE name space contains junctions similar to mount points in UNIX and NFS. These junctions allow the mounting of heterogeneous name spaces.
- File system mounting allow users to import files from remote server and share them. Spring naming service supports to create name spaces dynamically. It also supports to share individual naming context selectively.

6.8.3 Name Resolution

- Name resolution is iterative process. Example of iterative nature of resolution is use of aliases. If DNS server is requested to resolve an alias www.dcs.qmw.ac.uk, it first resolves the alias to another domain name *copper.dcs.qmw.ac.uk*, which must be further resolved to generate IP address. Hence, use of aliases presents cycles in name space in which case resolution never terminate. As a solution, threshold should be maintained and if resolution process crosses it then terminate the process. In other case administrator may carry out certain actions if aliases creates cycles.
- As DNS database is huge and used by large population. It is not stored on single server. Otherwise server would become bottleneck. Name services that are heavily used should use replication to ensure high availability. Administrative authority of the domain manages data belonging to domain which is stored on local name server. A name server may store data belonging to more than one domain. Local name server also needs to take help of other name servers to resolve the names as all the enquiries cannot be answered by it.
- Process of locating naming data from among several name servers so as to resolve the name is called navigation. The client name resolution software carries out navigation. DNS carries out iterative navigation as shown in Fig. 6.8.1.

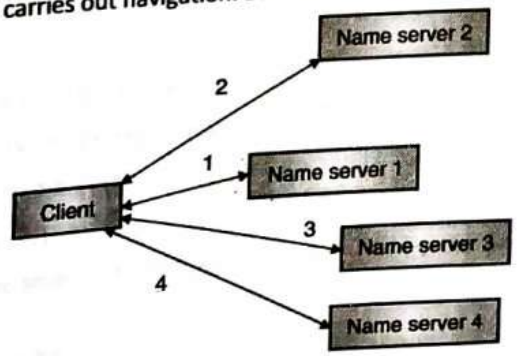
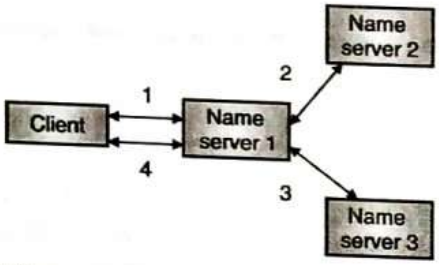
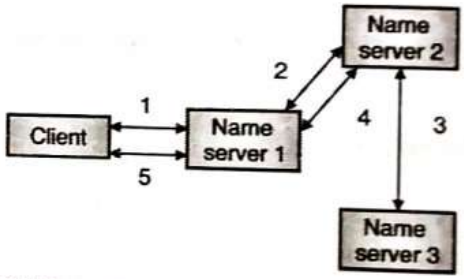


Fig. 6.8.1 : Client iteratively contact name server 1 to 4 in order to resolve name.

- Initially client presents name to local name server. If it has the name, it returns it immediately. Otherwise it will suggest another server which can help. Now resolution proceeds at new server. If needed, further navigation is carried out until the name is located or discovered to be unbound. In multicast navigation client multicast name to be resolved and needed object type to group of servers. Server storing this named attribute only replies to the request.
- In recursive name resolution, name server coordinates the resolution of name and returns result back to user agent. It is further classified as non-recursive and recursive server controlled navigation. In non-recursive server controlled navigation, client can choose any name server.
- This server then either multicast request to its peers or it communicates iteratively with peers. In recursive server controlled navigation, client once more contact single name server.
- If this server does not hold name then it contact to its peer that holds larger prefix of the name, which in turn attempt to resolve it. It is repeated until name is resolved. Fig. 6.8.2 shows non-recursive and recursive server controlled navigation.



(a) Non-recursive server controlled Navigation



(b) Recursive server controlled Navigation

Fig. 6.8.2

- The access to name server running in one administrative domain by client running in another administrative domain should be prohibited. In DNS and other name services, client name resolution software and server maintains **cache** of results of previous name resolution. On a clients request to resolve name, first client name resolution software enquire cache. If recent result of previous name resolution is found then return it to client. Otherwise request is forwarded to server. That server in turn may return result cached from other server.
- Caching improves performance and availability. It saves communication cost and improves response time. High level name servers such as root servers are eliminated due to caching.

6.8.4 Domain Name System

- DNS naming database is used across the internet. The objects named by DNS are computers. For these computers IP addresses are stored as attributes. In DNS domain names simply are called as domains. Internet DNS has bound millions of names. The look ups against these names are carried out from all around the world.
- Mainly DNS is used for naming across the Internet. DNS name space is partitioned organizationally and as per geography as well. In name highest level domain is mentioned at right. Following are the generic domains.
  - ***com :*** Commercial organizations
  - ***edu :*** Educational institutions and universities
  - ***gov :*** US Governmental agencies

- ***mil :*** US military Organizations
- ***net :*** major network support centers
- ***org :*** Organizations not listed in above domains
- ***int :*** International organizations

- Every country has its own domain. Some of the examples are :
  - ***us:*** United states
  - ***fr:*** France
  - ***in:*** India
- Countries other than USA use their own domain. For example, India has domain *ac.in* which corresponds to *edu* domain.

DNS Queries

Internet DNS is used for simple host name resolution and looking up electronic mail host :

- **Host name resolution :** In this, applications use DNS to resolve host names in to IP addresses. URL contains domain name. When it is given to browser, it makes DNS enquiry and obtains IP address. Browsers use http to communicate with web servers at an IP address with a reserved port number.
- **Locating the mail host :** Electronic mail software uses DNS to resolve domain names in to IP addresses of mail host.

DNS Name Servers, Navigation and Query Processing

- Scaling is achieved with combination of partitioning the naming database and by replicating and caching the part of this database close to the point of requirement. DNS database is distributed across logical network of servers. Each server holds part of database mainly of local domain so that local request can be fulfilled locally within the domain.
- Each server also records domain names and addresses of other servers. DNS naming data is divided into zones where each zone contains following data.
  - It contains attribute data for names in domains and less the data about sub-domains. For example, it could contain data for organization (*college.ac.in*) and less the data about department (*department.college.ac.in*).
  - Names and addresses of at least two servers in zone which maintains trustworthy and reliable data for the zone.
  - Names of name servers that holds data for delegated sub-domains and data which gives IP addresses of these servers quickly.
  - Parameters related to zone management that governs the replication and caching.
- Any server can cache data from other servers so that it can be required in future name resolution process. Each entry in zone contains time to live value so that cached data from un-authoritative server by client will remain useful. In this case, this un-authoritative server caches data from authoritative server. If client sends query after time to live period then un-authoritative server contact again to authoritative server. This minimizes network traffic and offers flexibility to system administrators.
- DNS can be used to store arbitrary attributes. Type of query specifies what is required. It can be IP address, name server, mail host or other information.

- A DNS client is resolver which is implemented in library software. It is simplest request–reply protocol. Both iterative and recursive resolution is supported by DNS and client side software specifies the type of resolution to be carried out.
- Name servers store the zone data in files in the form of resource records. Following are some examples of resource records for Internet database.

Type of Record	Associated Entity	Contents	Meaning
A	Host	IP address	IP address of this computer
MX	Domain	List of <preference, host pair>	Refers to mail server to handle mail addressed to this node
PTR	Host	Domain name	Holds canonical name of host
TXT	Any kind	Arbitrary text	Text string
SOA	Zone	Parameters governing zone	Holds information on represented zone
NS	Zone	Machine architecture and operating system	Holds information on the host this node represents.
CNAME	node	Domain name for alias	Symbolic link.

- Berkeley Internet Name Domain (BIND) is an implementation of DNS for machines having UNIS OS running on them. Client programs link in library software as the resolver. DNS name servers run named daemon. BIND permits three servers which are primary, secondary and caching-only servers. The named program implements just one of these as per configuration file contents.
- Typically, organization has one primary, one or more secondary that offer service for name serving on different LANs at the site. In addition to this, caching-only servers are run by individual machines to minimize network traffic and speed up the response time.

6.9 Directory Services

- Directory services are attribute based naming systems. Directory service stores binding between names and attributes and that look up entries matches with attribute based specifications called as directory service. Some of the examples are LDAP, X.500 and Microsoft's Active Directory Services. Directory service returns attributes of any objects. Attributes are more powerful compared to names. For example, Programs usually can be written to select objects by their attributes and not names.
- A discovery service is a directory service. This discovery service registers services provided in spontaneous networking. In spontaneous networking devices gets connected at any moment of time without any warning. There is no any administrative preparation carried out for these devices when they connect in network. It is required to support set of clients and services to be registered transparently. There should not be any human intervention for the same.
- To support this, discovery service offers interface for registering and de-registering these services automatically. It also offers interface for the clients to look up these services. For example, customer in hotel should be able to connect his laptop to printer automatically without configuring it manually.

- This is possible if laptop uses look up interface of discovery service that finds available network printers that fulfills users need. In this case required attributes of the printing service may specify whether "laser" or "inkjet", offers color printing or not, location of printer etc.
- Jini system is developed for spontaneous networking. It assumes JVM runs in all the machines and machines communicate with each other through remote method invocation. It offers facility to discover the service, for transaction, for shared da places, and for events. Discovery related components in Jini system are look up services, Jini clients, and Jini services. Jini client uses look up service. To locate the look up service, client multicast to well known IP multicast address. Look up services listen on a socket bound to the same address in order to receive such requests.

6.10 The Global Name Service (GNS)

- Global Name Service (GNS) was designed at DEC Systems Research Center. It offers facility for resource location, authentication and mail addressing. Following are the design goals of GNS.
  - To handle arbitrary number of names and to serve arbitrary number of organizations.
  - A long life time.
  - High Availability.
  - Fault isolation: Local failure does not affect the entire system.
  - Tolerance of mistrust.
- These goals indicate that any number of computers, users can be added in system and hence, it considered the support for scalability. If organizational structure changes then the structure of name space may also change. The service should also accommodate the changes in the names of individuals, organization etc.
- Considering the large size of distributed system and naming database, caching is used in GNS. It assumes that, changes in database will occur infrequently and hence slow propagation of updates is adopted. Client can detect and recover from use of out of date naming data.
- GNS naming database is composed of tree of directories which holds names and value. Path names are used to refer the directory similar to UNIX file system. Each directory is assigned with unique directory identifier (DI). A directory holds list of names and references. The leaves of directory tress also hold values which are further organized in value tree.
- In GNS, names consist of two parts : <directory name, value name>. First part identifies directory and second refers to the value tree. The attributes of user Rajesh in directory C would be stored in value tree named as <DI of Root directory/A/B/C, Rajesh>. The password in value tree can be referenced as :

  <DI of Root directory/A/B/C, Rajesh/password>
- The directory tree is partitioned and stored in many servers. Each partition is again replicated to many servers. If two or more concurrent updates take place then consistency of tree is maintained. Two directory trees can be merged by inserting new root above roots of these two trees to be merged. This problem is solved by unique directory identifier. Whenever client uses the name </A/B/C, Rajesh>, local user agent add prefix to the directory as <#567/A/B/C, Rajesh>.
- User agent then sends this derived name to GNS server. In the same way, user agent deals with relative name referring to working directories.

- GNS maintains table of well-known directories in which it list all the directories which are used as working roots. This table is held in current real root directory of the naming database. Whenever real root of naming data base changes due to adding the new root, all GNS servers are informed about location of new root. In GNS restructuring of database can be done if any organizational changes occurs.

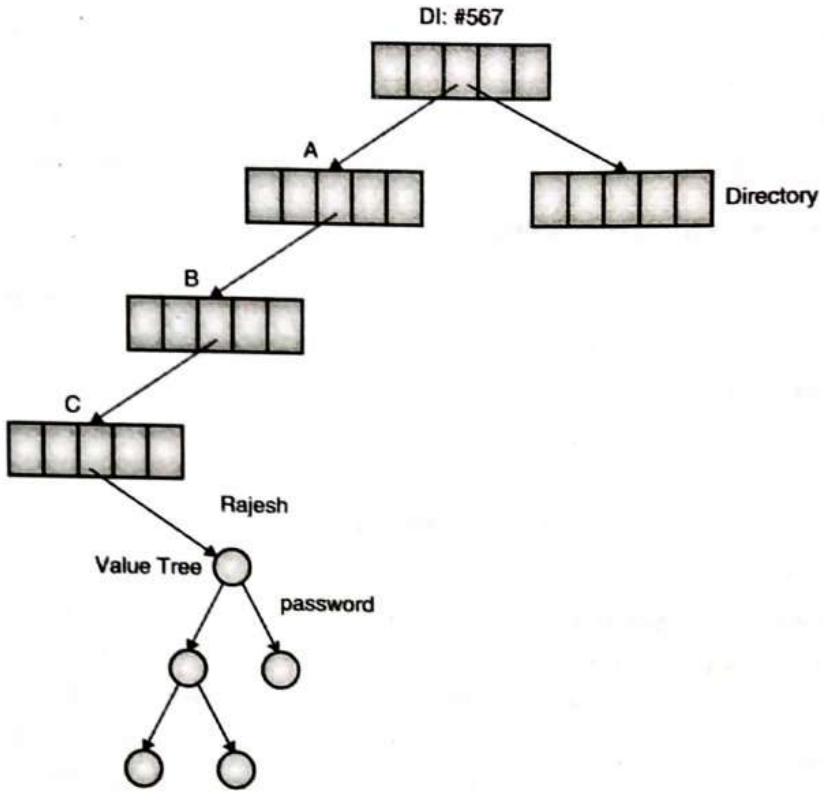


Fig. 6.10.1 : GNS directory tree

6.11 The X.500 Directory Service

- X.500 Directory Service is used to satisfy descriptive queries to look up names and attributes of other users or system resources. The use such service is quite diverse. For example, the enquiries can be for accessing white pages to obtain user's email address or yellow pages query may be for obtaining names and telephone numbers of garages.
- Individuals or organizations can use directory service to provide information about them and resources they wish to offer for use in network. It is possible for users to search directory for information with partial knowledge of its name, structure and contents. ITU and ISO standard organization have defined X.500 Directory Service as a network service.
- It is used for access to hardware and software services and devices. The X.500 servers maintain the data in tree structure with named node just like other name servers. Each node of tree in X.500, stores wide range of attributes. The entries can be searched by any combination of attributes.
- The X.500 name tree is called as **Directory Information Tree (DIT)**. The entire directory structure along with data associated with nodes is called as **Directory Information Base (DIB)**.

- It is intended to have single integrated DIB with information provided by organizations throughout the world. The portion of DIB is located in individual X.500 servers. Clients establish connection with server to access the directory by issuing the requests.
- Client can contact any server. If data are not in the part of DIB of contacted server then it will either invoke other servers or redirects the client to another server. In X.500, servers are called as Directory Service Agents (DSAs) and clients are called as Directory User Agents (DUAs). Following Fig. 6.11.1 shows the service architecture of X.500. It is one of the navigation model in which each DUA client interact with single DSU server which further accesses the other DSU servers to satisfy the client requests.

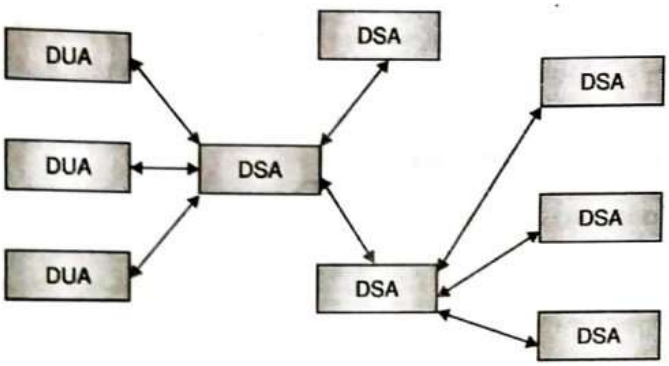


Fig. 6.11.1 : X.500 Service Architecture

- Each entry in DIB consists of name and attribute. The name of an entry corresponds to path from root to entry in DIT. Absolute or relative path names can be used. Also DSU can establish a context that includes base node and then use shorter relative names that gives path from base node to relative entry.
- Directory can be accessed by using two access requests : read and search.
  - ***read :*** An absolute or relative name to an entry is given along with a list of attributes to be read. DSA then navigates DIT to locate the entry. If the part of tree that includes entry is not available on this DSA server, then it passes request to other DSA servers. The retrieved attributes then it returns to client.
  - ***Search :*** This access request is attribute-based, base name and filter expression are supplied as arguments. The base name indicates the node in DIT from which search is to begin and filter expression is to be evaluated for every node below the base node. The search criterion is specified by filter. The search command then returns names for all entries for which filter evaluates TRUE value.
- DSA interface offers operations to add, delete and modify the entries. Access control is also provided for both query and update operation. As a result, access to some part of DIT can be restricted to user or group of users.

6.12 Designing Distributed Systems : Google Case Study

6.12.1 Google Search Engine

- From a distributed systems viewpoint, Google offers attractive extremely challenging requirements, mainly in terms of scalability, reliability, performance and openness. Like any web search engine, Google search engine return an ordered list of the most relevant results that match to the given query by searching the content of the Web. The search engine contains a set of services for crawling the Web and indexing and ranking the searched pages.

- The **crawler** locates and retrieves the contents of the Web and passes the contents onto the indexing subsystem. This is carried out by a software service called ***Googlebot***, which recursively reads a given web page, harvesting all the links from that web page and then scheduling further crawling operations for the harvested links.
- The **indexing** produces an index for the contents of the Web which is on a much larger scale. More specifically, indexing produces an ***inverted index*** mapping words in web pages and other textual web resources onto the positions where they occur in documents, including the accurate position in the document and other related information for example, the font size and capitalization. The index is also sorted to support efficient queries for words against locations.
- Indexing does not provide information about the relative importance of the web pages that contains a particular set of keywords. In **ranking**, higher rank denotes importance of a page and it is used to make sure that important pages are returned nearer to the top of the list of results than lower-ranked pages. Ranking in Google also considers factors like nearness of keywords on a page and their font size (large or small).

6.12.2 Google Applications and Services

- Apart from the search engine, Google now offers a wide range of web-based applications that also includes Google Apps. Now days, cloud computing also extensively supported by Google. It provides software as a service (SAS), which offers application-level software over the Internet, as web applications. A Google Apps is the main example of it. Google Apps includes set of web based applications such as Gmail, Google Docs, Google Sites, Google Talk and Google Calendar.
- Apart from SAS, with new addition of Google App Engine, Google now offers its distributed systems infrastructure as a cloud service. This cloud infrastructure supports all its applications and services, including its web search engine. The Google App Engine now offers external access to a part of this infrastructure, permitting other organizations to run their own web applications on the Google platform. Following are the examples of Google applications :
  - **Gmail :** It is a mail system of Google that hosts messages.
  - **Google Docs :** It is Web-based office suite which supports for editing of documents held on Google servers in shared mode.
  - **Google Sites :** These are Wiki-like web sites having shared editing facilities.
  - **Google Talk :** It offers instant text messaging and Voice over IP.
  - **Google Calendar :** It is Web-based calendar having all data hosted on Google servers.
  - **Google Wave :** It is a collaboration tool integrating email, instant messaging, wikis and social networks.
  - **Google News :** It is automated news aggregator site.
  - **Google Maps :** This App offers scalable web-based world map including high-resolution imagery and unlimited user generated overlays.
  - **Google Earth :** This App offers scalable near-3D view of the globe with unlimited user-generated overlays.
  - **Google App Engine :** It offers distributed infrastructure of Google to other parties outside of Google.
- Google infrastructure provides scalability and it uses approaches to scale in large scale. Google considers scalability problem related to dealing with more data, more queries, and expecting better results.

- Google provides high reliability for the search functionality and Apps it offers to customer. Its service level agreement contains 99.9% guarantees to offer services that includes all the Google applications.
- Google aims for low latency to user interactions to gain high **performance**. Google keeps the target of completing web search operations in 0.2 seconds and achieving the throughput to react to all incoming requests while handling very large datasets. Google also highly supports for **openness** in order to accommodate the wide range of web applications to be developed in future.
- To satisfy above stated requirements, Google has developed the overall system architecture as shown in 6.12.1. As shown in Fig. 6.12.1, the underlying computing platform is at the bottom and the well-known Google services and applications are at the top. The distributed infrastructures offering middleware support for search and cloud computing is provided by middle layer. This infrastructure offers the common distributed system services for developers of Google services and Apps and encapsulates key techniques that handle scalability, reliability and performance.
- This infrastructure bootstraps the development of new applications and services through reuse of the underlying system services and, more subtly, offers largely coherence to the growing Google code base by implementing common strategies and design principles.

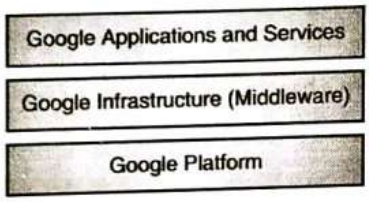


Fig. 6.12.1 : The Overall System Architecture of Google

6.12.3 Google Infrastructure

- Following Fig. 6.12.2 shows Google infrastructure. The system is constructed as a set of distributed services offering core functionality to developers. The *protocol buffers* component offers a common serialization format for Google, together with the serialization of requests and replies in remote invocation. Google *publish-subscribe* service supports the efficient dissemination of events to potentially large numbers of subscribers.

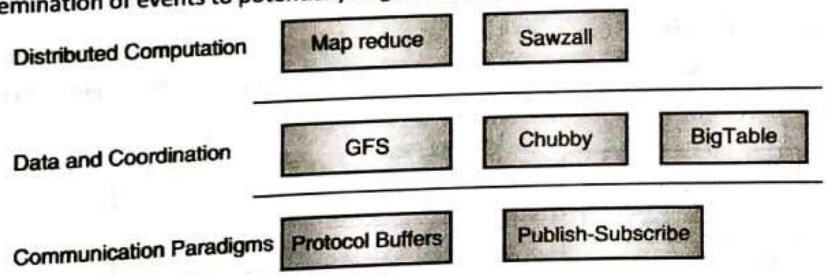


Fig. 6.12.2 : Google Infrastructure

- Data and coordination services offers unstructured and semi-structured\ abstractions for the storage of data coupled with services to support coordinated access to the data : GFS offers a distributed file system optimized for the particular requirements of Google applications and services. Chubby supports coordination services and the ability to store small volumes of data. Bigtable provides a distributed database offering access to semi-structured data.
- Distributed computation services offers means for carrying out parallel and distributed computation over the physical infrastructure : MapReduce supports distributed computation over potentially very large datasets.

- Sawzall provides a higher-level language for the execution of such distributed computations. Communication is supported through RMI, RPC and publish-subscribe service.

6.12.4 The Google File System (GFS)

- The GFS mainly aims at demanding and rapidly growing needs of Google's search engine and the Google web applications. Following are the requirements for GFS :
  - GFS must run reliably on the physical architecture.
  - GFS is optimized for the patterns of usage within Google, both in terms of the types of files stored and the patterns of access to those files.
  - GFS must fulfill all the requirements for the Google infrastructure as a whole.
- GFS provides a conventional file system interface offering a hierarchical namespace with individual files identified by pathnames. Following file operations are supported. The main GFS operations are very similar to those for the flat file service.
  - ***create*** – create a new instance of a file;
  - ***delete*** – delete an instance of a file;
  - ***open*** – open a named file and return a handle;
  - ***close*** – close a given file specified by a handle;
  - ***read*** – read data from a specified file;
  - ***write*** – write data to a specified file.
- The parameter for GFS *read* and *writes* operations specify a starting offset within the file. The API offers, *snapshot* and *record append* operations. The snapshot operation offers an efficient mechanism to make a copy of a particular file or directory tree structure. The *record append* operation supports the common access pattern whereby multiple clients carry out concurrent appends to a given file.

GFS Architecture

- Fig. 6.12.3 shows overall GFS architecture. In GFS, the storage of files is in fixed-size *chunks*, where each chunk is 64 megabytes in size.This size chosen is very large compared to other file system. It offers highly efficient sequential reads and appends of large amounts of data.

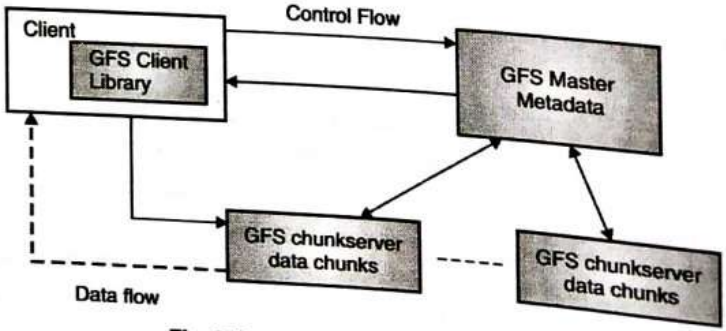


Fig. 6.12.3 : Overall GFS Architecture

- GFS provides a mapping from files to chunks and also supports standard operations on files, mapping down to operations on individual chunks. Figure shows an instance of a GFS file system as it maps onto a given physical cluster. Each GFS cluster has a single *master* and multiple *chunkservers* which jointly offer a file service to large numbers of clients concurrently accessing the data.
- The master manages metadata about the file system. It defines namespace for files, access control information and the mapping of each particular file to the associated set of chunks. All chunks are replicated and location of the replicas is maintained in the master. Replication provides reliability in case if any software or hardware failure occurs. The key metadata is maintained persistently in an operation log. This helps in recovery in the event of crashes.
- The master is centralized which can be single point of failure. The operations log is replicated on several remote machines, so the master can be restored on failure. GFS avoids heavy use of caching and clients do no cache file data. The location information of chunks is cached at clients when first accessed, in order to reduce interactions with the master. GFS does not have any strategy for server-side caching. It uses buffer cache in Linux to maintain frequently accessed data in memory.

Review Questions

Q. 1 Explain in short services provided by distributed file system.

Q. 2 Explain desirable features of good distributed file system.

Q. 3 Explain different file models.

Q. 4 Explain different file accessing models..

Q. 5 Explain different techniques to transfer data between client and server.

Q. 6 Explain different policies to propagate modifications.

Q. 7 Explain different cache validation schemes.

Q. 8 Differentiate between caching and remote service.

Q. 9 Differentiate between caching and replication.

Q. 10 What are the advantages of replication? Explain.

Q. 11 Explain in detail replication transparency.

Q. 12 Explain different approaches to update the multiple copies of file.

Q. 13 Explain quorum based protocol for updating of multiple copies.

Q. 14 Explain in detail network file system (NFS).

Q. 15 Explain architecture of NFS.

Q. 16 Explain in detail Andrew File System (AFS).

Q. 17 What are names, identifiers and addresses? Explain.

Q. 18 Write short note on "Name Spaces".

Q. 19 Explain different name resolution techniques.

Q. 20 Write note on domain name system.

Q. 21 Write note on directory services.

Q. 22 Write note on X.500 Directory Service.

Q. 23 Write note on global name service.

Q. 24 Explain Google applications and services.

Q. 25 Explain the overall system architecture of Google.

Q. 26 Explain Google file system.

□□□