CONTENTS

CHAPTER - 1

INTRODUCTION TO DISTRIBUTED SYSTEM

CHAPTER - 1

INTRODUCTION TO DISTRIBUTED SYSTEM

1.1 Introduction to Distributed System

A *distributed system* is one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. In other words, a *distributed system* consists of a collection of autonomous computers, connected through a network and distributed operating system software, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

Each computer in a distributed system have their own memory in contrast to a parallel system where all the processors have access to common memory for information exchange. The main aim is to distribute the system over more than one physical locations so as to overcome the operational and organizational issues of single server system.

Distributed systems are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks, all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading "distributed systems".

1.1.1 Architecture of Distributed System

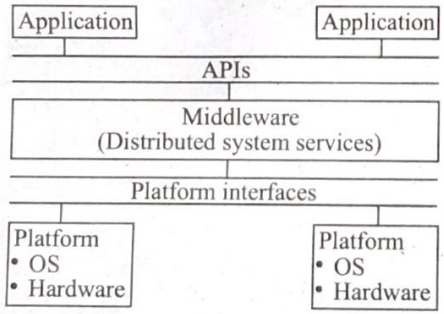


Fig.: Architecture of distributed system.

The architecture of a system is its structure in terms of separately specified components and their interrelationship. In architecture, there are mainly 3 layers. They are:

i. Hardware

ii. Software

iii. Middleware

Autonomous applications are stored in independent computers. The middleware provides the distributed services assuring the users have the same views of the system. The middleware acts as a communication channel between the application or the users.

1.2 Examples of Distributed System

i. The Internet

The internet is a global system of interconnected computer network that use the standardized Internet Protocol Suite(TCP/IP) to serve billions of users worldwide. It is a networks that consists of millions of private and public, academic, business and government networks of local to global scope that are linked by copper wires, fiber-optic cables, wireless connection, and other technologies.

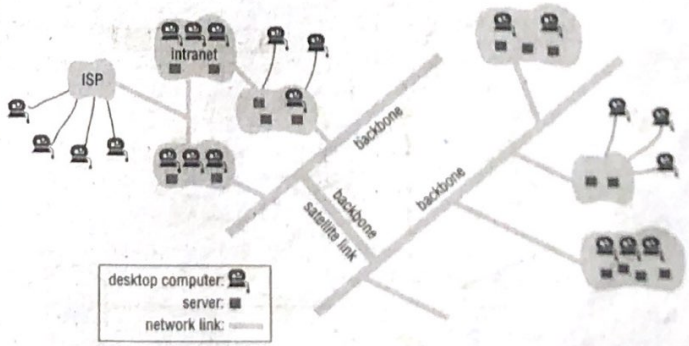


Fig.: A typical portion of the Internet.

Figure above illustrates a typical portion of the Internet. Programs running on the computers connected to it interact by passing messages, employing a common means of communication. The Internet is also a very large distributed system. It enables users, wherever they are, to make use of services such as the World Wide Web, email, and file transfer. The set of services is open-ended it can be extended by the addition of server computers and new types of service.

ii. Bit Torrent

Bit Torrent is a protocol for file sharing across the web. The aim of Bit Torrent is to facilitate file transfer between different peers in the network without having to go through main server. A Bit Torrent client software allows connection of multiple computers across the world to download a file. When a torrent file is opened through a torrent client, the computer connects to a tracker, which acts as a coordinator machine. Tracker helps in peer discovery. Peers are the nodes in the network which contains the file that we want. Peers can be seeder or Leecher. Leecher is the user who is downloading a file. Seeder is the user who is uploading required file. From the user's perspective, the whole system looks like a single network that is serving the file transfer.

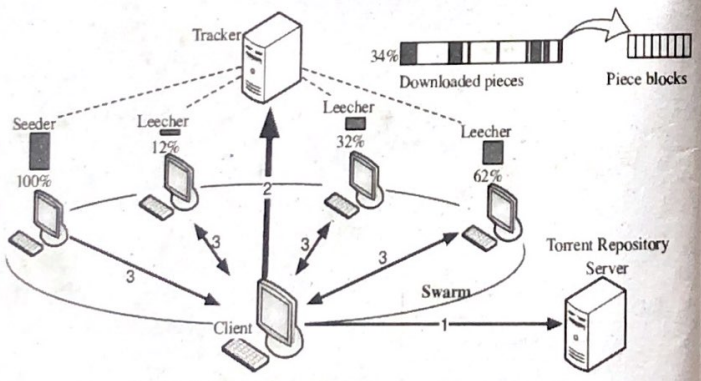


Fig.: Architecture of BitTorrent.

iii. Mobile and Ubiquitous Computing

Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- Laptop computers
- Handheld devices, including personal digital assistants (PDAs), mobile phones, GPS, pagers, video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

Broadly speaking, mobile computing is concerned with exploiting the connectedness of devices that move around in the everyday physical world; ubiquitous computing is about exploiting the increasing integration of computing devices with our everyday physical world.

iv. Intranets

An intranet is a portion of the Internet that is separately administered and has a boundary that can be configured to enforce local security policies. Figure below shows a typical intranet. It is composed of several local area networks (LANs) linked by backbone connections.

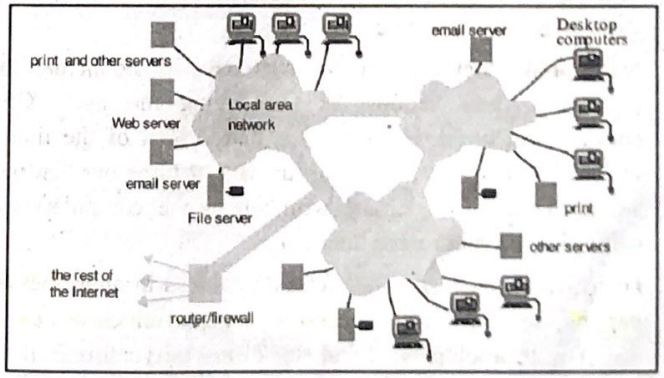


Fig : A Typical Intranet.

An intranet is connected to the internet via a router, which allows the users inside the intranet to make use of services elsewhere such as the Web or email. It also allows the users in other intranets to access the services it provides. An intranet is protected by a firewall against unauthorized access.

v. **Blockchain and Bitcoin**

vi. **Decentralized Authentication System (Single Sign On - SSO)**

vii. **Distributed Messaging (Amazon SQS, Rabbit MQ)**

1.3 Characteristics of Distributed System

Some of the characteristics of distributed system are explained as follows:

i. **Concurrency:** In a network of computers, concurrent program execution is the norm. I can do my work on any computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example, computers) to the network.

Concurrency allows programs that share resources to execute concurrently. It reduces latency and increase throughput.

Example: Web application is concurrent as it can be used by various users at the same time.

ii. **No global clock:** When progress need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs actions occur. But, it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clock.

Distributed system communicates messages through message passing, so there is no necessity of global clock system. In primitive technologies, global clock must besynchronized. But as there is no correct global timingsystem, it is difficult to synchronize.

iii. **Independent failure nodes:** Each machine has probability to fail or crash at any time. In centralized system, if server fails, the whole system crashes. In distributed system, failure of a single node does not hamper the entire system. This characteristic provides availability and reliability.

1.4 Advantages and Disadvantages of Distributed System

Advantages of distributed system:

- Better price/performance ratio (cost effective way to increase computing power)
- Offers high availability (system does not fail even a single computer crashes)
- Efficient speed due to load and work distributing
- Scalability (computing power is modular)
- Data sharing (allows users to share data using common database)

Disadvantages of distributed system:

- Difficult to implement
- Exchange of information between components require coordination creating processing overheads

1.5 Design Issues/ Goals/Challenges of Distributed System

The construction of distributed systems produces many challenges. These are:

i. **Heterogeneity**

Distributed systems must be constructed from a variety of different networks, operating systems, computer hardware, and programming languages. The Internet communication protocols mask the difference in networks, and middleware can deal with the other differences.

ii. **Openness**

The openness is the characteristic that determines whether the system can be extended or re-implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.

Distributed systems should be extensible the first step is to publish the interfaces of the components, but the integration of components written by different programmers is a real challenge.

iii. **Security**

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components:

- Confidentiality
  - Protection against disclosure to unauthorized individuals.
- Integrity
  - Protection against alteration or corruption.
- Availability
  - Protection against interference with the means to access the resources. Encryption can be used to provide adequate protection of shared resources and to keep sensitive information secret when it is transmitted in messages over a network. Denial of service attacks are still a problem.

iv. **Scalability**

Distribute systems operate effectively and efficiently at many different scales, ranging from a small intranet to the Internet. A system is described as scalable if it will remain effective when there is a significant increase in the number of resources and the number of users.

A distributed system is scalable if the cost of adding a user is a constant amount in terms of the resources that must be added. The algorithms used to access shared data should avoid performance bottlenecks and data should be structured hierarchically to get the best access times. Frequently accessed data can be replicated.

v. **Failure handling**

Any process, computer or network may fail independently of the others. Therefore each component needs to be aware of the possible ways in which the components it depends on may fail and be designed to deal with each of those failures appropriately.

vi. **Concurrency**

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. Any object that represents a shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment.

For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

vii. **Transparency**

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. The implications of transparency are a major influence on the design of the system software.

The application process and resources of distributed system are physically distributed across multiple computers. The system should act as a single coherent system to its users. Transparency indicates hiding the fact of distributed nature to provide coherent interface to the user.

Layers of transparency:

a. Access transparency

Access transparency enables local and remote resources to be accessed using identical operations.

To explain access transparency, consider a graphical user interface with folders, which is the same whether the files inside the folder are local or remote. Another example is an API for files that uses the same operations to access both local and remote files.

b. Location transparency

Location transparency enables resources to be accessed without knowledge of their physical or network location. Users cannot tell where hardware and software resources (CPUs, files, database) are located; the name of the resources, should not encode the location of the resources.

c. Migration (mobility) transparency

Mobility transparency allows the movement of resources and clients within a system without affecting the operation of users or programs.

To illustrate mobility transparency, consider the case of mobile phones. Suppose that both caller and callee are travelling by train in different parts of a country, moving from one environment (cell) to another. We regard the caller's phone as the client and the callee's phone as a resource. The two phone users making the call are unaware of the mobility of the phones (the client and the resource) between cells.

d. Replication transparency

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

e. Concurrency transparency

Concurrency transparency enables several processes to operate concurrently using shared resources without interference between them.

f. Failure transparency

Failure transparency enables the concealment of faults, allowing users and application program to complete their tasks despite the failure of hardware or software components.

To example failure transparency, consider an electronic mail, which is eventually delivered, even when servers or communication links fail. The faults are masked by attempting to retransmit messages until they are successfully delivered, even if it takes several days. Middleware generally converts the failures of networks and processes into programming-level exceptions.

g. Performance transparency

Performance transparency allows the system to be reconfigured to improve performance as loads vary.

1.6 Models of Distributed System

1. Architectural Models

The architecture of a system is its structure in terms of separately specified components. The major goal is to ensure that the structure will meet present and likely future demands on it.

An *architectural models* of a distributed system is concerned with the placements of its parts and the relationships between

them. Most distributed system are arranged according to one of a variety of architectural model. Example: Client-server model, peer-to-peer model.

i. Client-Server Model

Client-server model is the most important and most widely distributed system architecture. The client-server model is prevalent – the web and other Internet services such as ftp, news and mail as well as web services and the DNS are based on this model, as are filing and other local services. Services such as the DNS that have large number of users and manage a great deal of information are based on multiple servers and use data partition and replication to enhance availability and fault tolerance. Caching by clients and proxy servers is widely used to enhance the performance of a service.

Client processes interact with individual server processes in a separate host computers in order to access the shared resources. Client and server roles are assigned and changeable.

- Server may in turn be clients of other servers example: a web server is often a client of a local file server that manages the file in which the web pages are stored.
- Servers partition the set of objects on which the services is based and distribute them among themselves (web data and web servers). Servers maintain replication copies of the services objects on several hosts for reliability and performance (example: Alta vista).

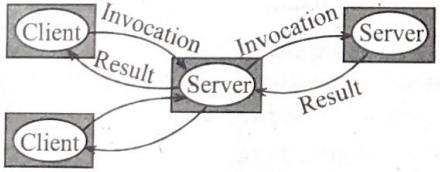


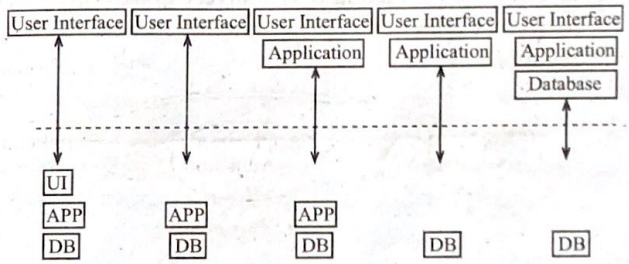
Fig: Client invoice individual servers.

ii. Multitiered Architectures

- **Single-tiered**: dumb terminal/mainframe configuration
- **Two- tiered**: Client/single server configuration
- **Three- tiered**: each layer on separate machine

The simplest organization is to have only two types of machine.

- A client machine containing only the programs implementing (part of) the user- interface level.
- A server machine containing the rest.



Proxy servers and caches

A cache is a store of recently used data objects that is closer than object itself. When an new object is received at a computer it is added to the cache store, replacing some existing objects if necessary. When an object is needed by a client process that caching services checks the cache and supplies the object from there in case of an up-to-date copy is available. If not, an up-to-date copy is fetched. Caches may be collocated with each client or they may be located in a proxy server that can be shared by several clients.

Proxy servers provides a shared cache of web resources for client machines at a site or across several site. It increases availability and performance of a services by reducing load the WAN and web servers. It may be used to access remote web servers through a firewall.

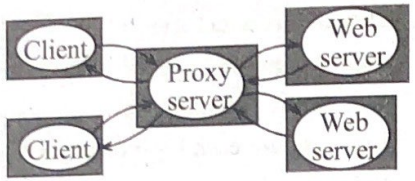


Fig: Web proxy

Peer processes

All processes play similar roles without destination as a client or a server. Interacting cooperatively to perform a distributed activity. Communication pattern will depend on application requirement.

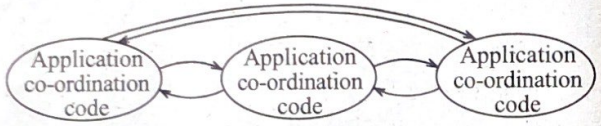


Fig.: Peer processes

iii. Peer-to-peer (P2P) architecture

In this architecture, all of the process play similar roles, interacting cooperatively as peers to perform a distributed activities or computation without any distinction between client s and servers or the computers that they run on. It includes the exchange of information processing cycles, cache storage and disk storage for files.

The aim of the peer-to-peer architecture is to exploit the resources (both hardware and data) in a large number of participating computers for the fulfillment of a given task or activity.

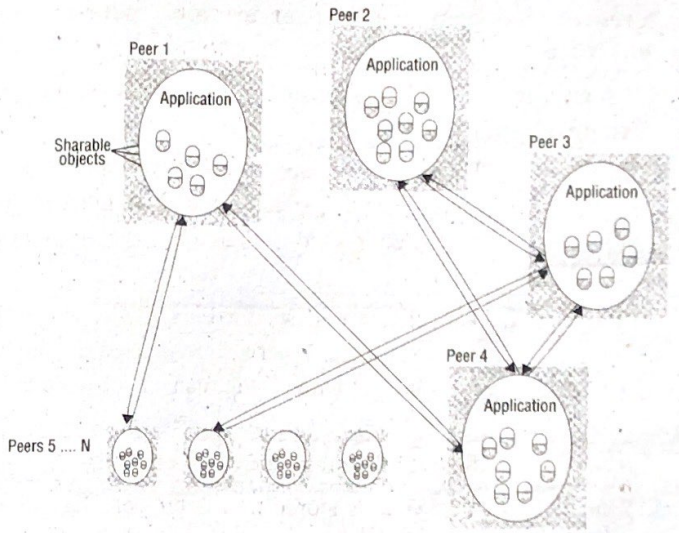


Fig.: A distributed application based on the peer-to-peer architecture

The figure illustrates the form of a peer-to-peer application. Application are composed of large numbers of peer processes running on separate computers and the pattern of communication between them depends entirely on application requirement. A large number of data objects are shared, an individual computer holds only a small part of the application database, and the storage, processing and communication loads for access to objects are distributed across many computers and network links. Each object is replicated in several computers to further distribute the load and to provide resilience in the event of disconnection of individual computers (as is inevitable in the large, heterogeneous networks at which peer-to-peer systems are aimed). The need to place individual objects and retrieve them and to maintain replicas amongst many computers renders this architecture substantially more complex than the client-server architecture.

Differences between client-server and peer-to- peer architecture:

S.N.	Basis for comparison	Client-server	Peer-to-peer
1	Basic	There is a specific server and specific clients connected to the server	Client and servers are not distinguished; each node acts as client and server
2	Service	The client requests for service and server responds with the service.	Each node can request for services and can also provide the services.
3	Focus	Sharing the information	Connectivity
4	Data	The data is stored in a centralized server	Each peer has its own data
5	Server	When several clients request for the services simultaneously, a server can get bottlenecked	As services are provided by several services distributed in the P2P system, a server in not bottlenecked
6	Expense	The client server are expensive to implement	P2P are less expensive to implement
7	Stability	Client server is more stable and scalable	P2P suffers if the number of peer increase in the system.

2. Fundamental Models

Fundamental model of a system gives the formal definition of the system. In general, fundamental model provides operating system with the general ingredients that is necessary to understand system behavior. The purpose of a model is

- To make explicit all the relevant assumption about the systems we are modelling.

- To make generalizations concerning what is possible or impossible, given those assumptions. The generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees are dependent on logical analysis and, where appropriate, mathematical proof.

They are concerned with more formal description of the properties that are common in architectural model. It has 3 sub-part.

a. Interaction Model

The interaction model is concerned with the performance of processes and communication channels and the absence of a global clock. It identifies a synchronous system as one in which known bounds may be placed on process execution time, message delivery time and clock drift. It identifies an asynchronous system as one in which no bounds may be placed on process execution time, message delivery time and clock drift – which is a description of the behavior of the Internet.

b. Failure Model

The failure model classifies the failures of processes and basic communication channels in a distributed system. Masking is a technique by which a more reliable service is built from a less reliable one by masking some of the failures it exhibits. In particular, a reliable communication service can be built from a basic communication channel by masking its failures. For example, its omission failures may be masked by re-transmitting lost messages. Integrity is a property of reliable communication–it requires that a message received be identical to one that was sent and that no message be sent twice. Validity is another property–it requires that any message put in the outgoing buffer be delivered eventually to the incoming message buffer.

c. Security Model

The security model identifies the possible threats to processes and communication channels in an open distributed system. Some of those threats relate to integrity: malicious users may tamper with messages or replay them. Others threaten their privacy. Another security issue is the authentication of the principal (user or server) on whose behalf a message was sent. Secure channels use cryptographic techniques to ensure the integrity and privacy of messages and to authenticate pairs of communicating principals.

1.7 Types of Distributed System

There are 3 types of distributed systems

1. Distributed computing system
2. Distributed information system
3. Distributed pervasive system

1. Distributed computing system

Distributive computing systems are typically deployed for high-performance applications often originating from the field of parallel computing.

a. Cluster computing systems
b. Grid computing Systems
c. Cloud computing Systems

a. Cluster computing

The cluster computing that has high speed LAN, Each node runs the save operating system. It is used for parallel programming and users have impression that the processing is performed by a single resource.

The simplest architecture is described by symmetric cluster. Each node functions as an individual computer.

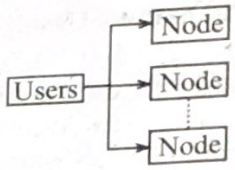


Fig.: Symmetric cluster

Also, the asymmetric cluster computer is a head node that acts as a gateway between other nodes and a gateway between other nodes and the others. All traffic must pass through heads providing high level of security.

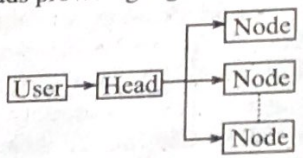


Fig.: Asymmetric cluster

b. Grid computing

Grid computing is a computer network in which each node's resources are shared with every other node, in the system. The nodes may be different in hardware, software and network technology. Every authorized node have access to enormous processing power and storage capacity. It works on the principle of pooled resources (share the load across multiple nodes to complete tasks more efficiently and effectively).

Architecture

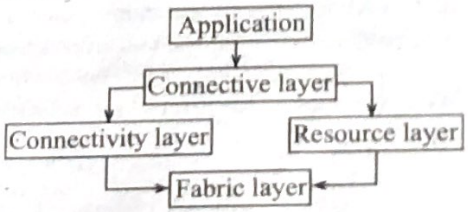


Fig: A layered architecture of grid computing.

The architecture consists of four layers.

Fabric layer:

The lowest fabric layer provides interfaces to local resources at a specific site. These interfaces are tailored to allow sharing of resources within a virtual organization. Typically, they will provide functions for querying the state and capabilities of a resource, along with functions for actual resource management (e.g., locking resources).

Connectivity layer:

It consists of communication protocols for supporting grid transactions that span the usage of multiple resources. For example, protocols are needed to transfer data between resources, or to simply access a resource from a remote location. In addition, the connectivity layer will contain security protocols to authenticate users and resources. In many cases human users are not authenticated; instead, programs acting on behalf of the users are authenticated. In this sense, delegating rights from a user to programs is an important function that needs to be supported in the connectivity layer. We return extensively to delegation when discussing security in distributed systems.

Resource layer:

It is responsible for managing a single resource. It uses the functions provided by the connectivity layer and calls directly the interfaces made available by the fabric layer. For example, this layer will offer functions for obtaining configuration information on a specific resource, or, in general, to perform specific operations such as creating a process or reading data. The resource layer is thus seen to be responsible for access control, and hence will rely on the authentication performed as part of the connectivity layer.

Collective layer:

It deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources,

data replication, and so on. Unlike the connectivity and resource layer, which consist of a relatively small, standard collection of protocols, the collective layer may consist of many different protocols for many different purposes, reflecting the broad spectrum of services it may offer to a virtual organization.

Application layer:

It consists of the applications that operate within a virtual organization and which make use of the grid computing environment.

c. Cloud Computing

Cloud computing is the on-demand delivery of compute power, database, storage, application and other IT resources via the Internet with pay-as-you-go pricing.

The characteristics of cloud computing are:

- **On-demand self-service**

  Cloud computing resources can be provisioned without human interaction from the service provider. In other words, a manufacturing organization can provision additional computing resources as needed without going through the cloud service provider.

- **Broad network access**

  Cloud computing resources are available over the network and can be accessed by diverse customer platforms. In other words, cloud services are available over a network– ideally high broadband communication link – such as the Internet, or in the case of a private clouds it could be a local area network (LAN).

- **Multi-tenancy and resource pooling**

  Cloud computing resources are designed to support a multi-tenant model. Multi-tenancy allows multiple customers to share the same applications or the same physical infrastructure while retaining privacy and security over their information.

- **Rapid elasticity and Scalability**

  Cloud computing resources can scale up or down rapidly and, in some cases, automatically, in response to business demands. The usage, capacity, and therefore

cost, can be scaled up or down with no additional contract or penalties.

- **Measured service**

  Cloud computing resources usage is metered and manufacturing organizations pay accordingly for what they have used.

Advantages:

- Affordable (pay per use model)
- Adaptable (owners control over core code)
- Multitenant (provide applications to multiple customers)
- Reliable
- Scalable
- Secure

Cloud computing service models:

- SaaS (Software as a Service)
- IaaS (Infrastructure as a Service)
- PaaS (Platform as a Service)

Cloud computing deployment models:

- Public clouds
- Private clouds
- Hybrid clouds

2. **Distributed Information System**

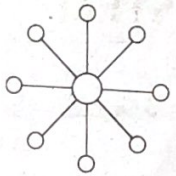
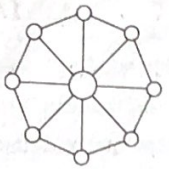
   It is used for distributing information across several servers. Communication models used are RPC and RMI and mostly associated with transaction processing system (TPS).

3. **Distributed Pervasive System**

   Distributed system involving mobile and embedded computer like wireless, PDA, sensor, smart phone and so on. It performs following tasks

   1. Auto connect to different network.
   2. Discover services and react accordingly
   3. Auto self configuration

Differences between centralized and distributed system:

Centralized system	Distributed system
1. All the computation in this system is done in one particular system.	The calculation and computation is distributed to multiple computers.
2. There is single point failure.	There is no single point failure.
3. They have global state.	They do not have global state.
4. The characteristics are:- Presence of a global clock, one single central unit, dependent failure of components.	The characteristics of distributed system are: concurrency of components, lack of global clock, independent failures of components.
5.  →Server/master →Computer/slave **Figure:** Centralized system visualization	5.  →Node/Computer **Figure:** Distributed system visualization
6. Advantages of centralized are: Easy to physically secure, smooth and elegant personal experience, dedicated resources, quick update are possible	6. Advantages of distributed system are: Higher performance than centralized system, higher reliability, easier to share data/resources

CHAPTER-2

DISTRIBUTED OBJECT AND FILE SYSTEM

CHAPTER-2

DISTRIBUTED OBJECT AND FILE SYSTEM

2.1 Introduction

2.1.1 Distributed Objects

In distributed computing, distributed objects are objects (in the sense of object-oriented programming) that are distributed across different address spaces, either in different processes on the same computer, or even in multiple computers connected via a network, but which work together by sharing data and invoking.

Objects that can receive remote method invocation are called remote objects and they implement a remote interface. Programming models for distributed application are:

1. **Remote procedure call (RPC)**

   The remote procedure call (RPC) approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local.

2. **Remote method invocation (RMI)**

   Remote method invocation (RMI) is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations.

3. **Event based distributed programming**

   Object receive asynchronous notifications of events s happening on remote computers/ processes.

2.2 Communication Between Distributed Objects

The object-based model for a distributed system extends the model supported by object-oriented programming languages to make it apply to distributed objects. This section addresses communication between distributed objects by means of RMI. The material is presented under the following headings:

- **The object model :**
  A brief review of the relevant aspects of the object model, suitable for the reader with a basic knowledge of an object-oriented programming language, for example Java or C++.
- **Distributed objects:**
  A presentation of object based distributed systems, which argues that the object is very appropriate for distributed system.
- **The distributed object model:**
  The distributed object model is an extension of the local object model used in object-based programming languages. Encapsulated objects form useful components in a distributed system, since encapsulation makes them entirely responsible for managing their own state, and local invocation of methods can be extended to remote invocation. Each object in a distributed system has a remote object reference (a globally unique identifier) and a remote interface that specifies which of its operations can be invoked remotely.
- **Design issues:**
  A set of arguments about the design alternatives:
  1. Local invocations are executed exactly once, but what suitable semantics is possible for remote invocations
  2. How can RMI semantics be made similar to those of local method invocation and what differences cannot be eliminated?

- **Implementation:**
  An explanation as to how a layer middleware above the request reply protocol may be designed to support RMI between application level distributed objects.
- **Distributed garbage collection:**
  A presentation of an algorithm for distributed garbage collection that is suitable for use with the RMI implementation.

2.3 Remote Procedure Call (RPC)

The concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing, with the goal of making the programming of distributed systems look similar, if not identical, to conventional programming—that is, achieving a high level of distribution transparency. In RPC, procedures on remote machines can be called as if they are procedures in the local address space. The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call.

RPC is a remote communication medium in which a client program calls a procedure in another program running in a server process.

- Client has a request message that RPC translate and sends to the server-The request may be procedure call or a function call to the remote server-client sends request with necessary arguments it's kernel. -The kernel validate the calls and copies arguments into the kernel buffer from client address space. The arguments are then passed to server's kernel space from client kernel space
- The call is made to the server by copying the arguments to server address space by the kernel

Fig.: Basic RPC model

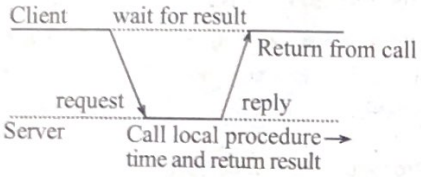


Fig.: RPC between client & server

Basic RPC operation:

1. Client procedure calls client stub.
2. Client stub build message and call local OS.
3. Client OS sends message to remote OS.
4. Remote OS gives message to server stub.
5. Server stub unpacks parameter and call server procedure.
6. Server executes and return result to server stub.
7. Server stub packs it in message and call local OS.
8. Server OS sender message to client OS.
9. Client OS gives message to client stub.
10. Stub unpacks result and return to client procedure client.

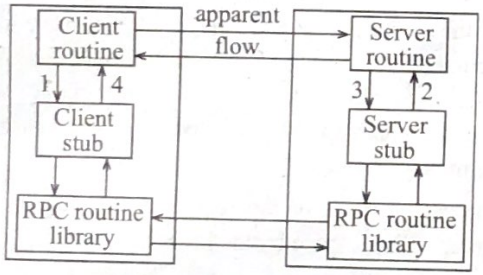


Fig.: RPC mechanism

Issues while performing RPC:

1. **Marshalling and unmarshalling**

   Marshaling is the process of passing the arguments into a message packet and Unmarshalling is the process of unpacking the arguments received from the call packet.

   Each type of parameter has their own representation. So, they must be known to the module.

2. **Semantics**

   Call by reference is not possible in RPC as the client and server do not share an address space.

3. **Binding**

   How does the client know who to call and where the service resides?

4. **Transport protocol**

   What protocol is used to transport the parameters?

5. **Exception handling**

   How are errors handled?

2.4 Remote Method Invocation (RMI)

Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects. In RMI, a calling object can invoke a method in a potentially remote object. As with RPC, the underlying details are generally hidden from the user.

The commonalities between RMI and RPC are as follows:

- They both support programming with interfaces, with the resultant benefits that stem from this approach.
- They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once.
- They both offer a similar level of transparency – that is, local and remote calls employ the same syntax but remote

interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

The following differences lead to added expressiveness when it comes to the programming of complex distributed applications and services.

- The programmer is able to use the full expressive power of object-oriented programming in the development of distributed systems software, including the use of objects, classes and inheritance, and can also employ related object-oriented design methodologies and associated tools.
- Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC

RMI is the method by which object is different processes can communicate with each other. It allows objects in one process to invoke or call methods of an objects in another process. Each process consists of set of objects. These objects may receive local invocation or remote invocation or both. object that receive remote. invocation is called remote objects.

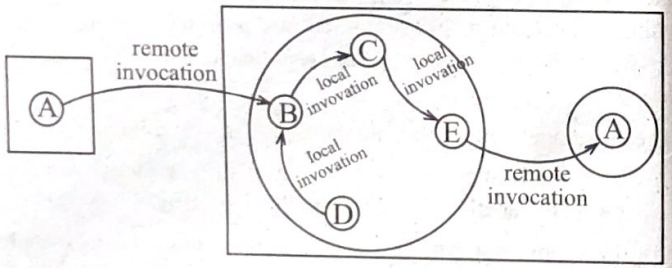


Fig.: Remote and local method invocation

For local invocation, it is possible only if the invoking object has reference to invoked object. Remote invocation is possible only

if the invoking object has access to remote object references of remote object. Each remote object has a remote interface, that contain which of the method of remote object can be invoked remotely by object from another processes.

Structure of Request-Reply message:

- Message type (int) - 0 : request, 1 : reply
- Request id (int)
- Object reference (remote object ref)
- Method id (int/method)
- Arguments (array of bytes)

Structure of remote object references:

32 bit	32 bit	32 bit	32 bit	
Internet address	port number	time	object number	interface remote object

Middleware implementations of RMI provide components (including proxies, skeletons and dispatchers) that hide the details of marshalling, message passing and locating remote objects from client and server programmers. These components can be generated by an interface compiler. Java RMI extends local invocation to remote invocation using the same syntax, but remote interfaces must be specified by extending an interface called Remote and making each method throw a Remote Exception. This ensures that programmers know when they make remote invocations or implement remote objects, enabling them to handle errors or to design objects suitable for concurrent access.

Issues while performing RMI:

1. **Invocation semantics**

   Invocation is done through request reply protocol, so it can be implemented as either:

   - Retry request message
   - Duplicate filtering
   - Retransmission of results

In local invocation, the semantics is exactly once.

RMI invocation semantics is produced by the choice request reply protocol that is implemented.

2. **Transparency**
   - Invoking object has reference to involved object (local method invocation)
   - Invoking object has access to remote object ref of remote object.

The major problems in maintaining transparency are:

- Different RMI semantics
- Susceptibility to failures
- Protection against interference in concurrent scenario.

2.5 Introduction to DFS

File system is important for organization, storage retrieval naming, sharing and protection of files. File contains data and attributes. DFS is a model offline system that is distributed across multiple machines or process. DFS provides file services to the client of distributed systems.

2.5.1 Distributed File System

A distributed file system (DFS) is simply a classical model of a file system distributed across multiple machines. The purpose is to promote sharing of dispersed file.

A file system provides a service for clients. The servers interface is the normal set of file operation create, read etc. on file.

A distributed file system enables programs to store and access remote file exactly as they do on local ones, allowing users to access from any computer on the internet advances in higher band width connectivity of switched local networks disk organization have lead high performance and highly scalable file system.

Requirements of DFS:

i. It must support transparency of access, location, migration, performance and so on.

ii. Concurrent file updates should be controlled.

iii. It support for file replication to enhance scalability and fault tolerance.

iv. Service interface should be defined such that it can be implemented in heterogeneous system.

v. It maintain file consistency

vi. It provide access control mechanism to enhance data security

Importance of DFS:

i. It supports sharing of information in the form of files throughout the intranet.

ii. It enables programs to store and access remote files exactly as they do an local ones .

Issues of DFS:

The key design issues for distributed file systems are:

- The effective use of client caching to achieve performance equal to or better than that of local file systems;
- The maintenance of consistency between multiple cached client copies of files when they are updated;
- Recovery after client or server failure;
- High throughput for reading and writing files of all sizes;
- Scalability.

2.6 File Service Architecture

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a flat file service, a directory service and a client module. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files. The client module provides a single programming interface with operations on files similar to

those found in conventional file systems. The design is open in the sense that different client modules can be used to implement different programming interfaces, simulating the file operations of a variety of different operating systems and optimizing the performance for different client and server hardware configurations.

The division of responsibilities between the modules can be defined as follows:

1. Flat file service
2. Directory file service
3. Client module

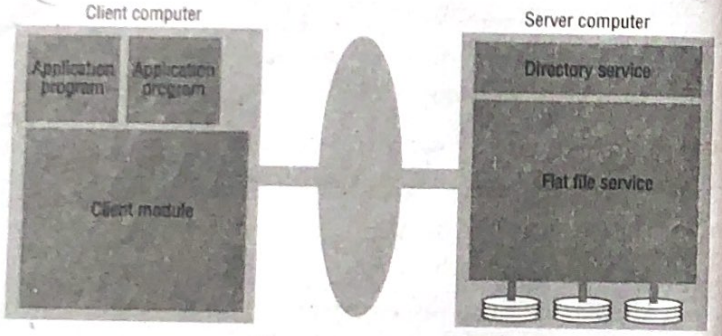


Fig: File service architecture

1. Flat File Service

The flat file service is concerned with implementing operations on the contents of files. Unique file identifiers (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

2. Directory File Service

The directory service provides a mapping between text names for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

3. Client Module

A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Stateful and Stateless Services in File System

- **Stateful**: A server keeps track of information about client's request. It maintain what files are opened by a client; connection identifiers; server caches. Memory must be reclaimed when client closes file or when client dies.
- **Stateless**: Each client request provides complete information needed by the server (i.e., filename, file offset. The server can maintain information on behalf of the client, but it's not required. The useful things to keep include file information for the N files touches.

2.7 Sun Network File System

Sun NFS is the first successful network file system. It is developed by Sun micro systems for their disk less workstations and focuses on transparency. It is designed for robustness and adequate performance.

Properties of Sun NFS:

1. It acts as both specification and implementations of how to access remote file.
2. It focuses on sharing a file system in transparency way.
3. I sues client server model. A node can act both as client as well as server.
4. It uses mount to make server file system visible for a client.
5. It is stateless.
6. It is machine and operating system independent

SUN network file system operation:

1. Search for file within directory
2. Read a set of directory entries
3. Manipulate links and directories
4. Read file attribute
5. Write file attribute
6. Read file data
7. Write file data

SUN network file system architecture

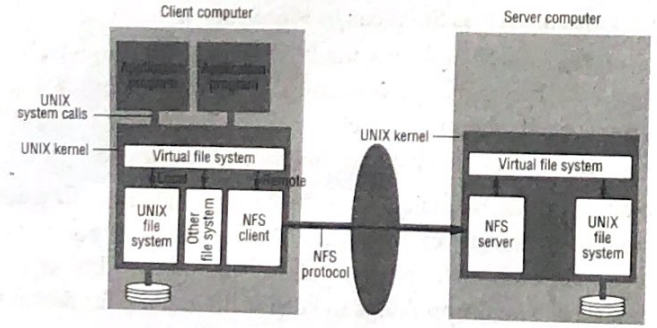


Fig.: SUN NFS architecture

1. **Protocol**
   - It uses SUN RPC mechanism and XDR standard.
   - The protocol is ~~stales~~ stateless that enhance crash recovery.
   - Each procedure call must contain all the information necessary to complete the call.
2. **Sever** Server
   It provides file handle consisting of
   - File system ID: Identify disk partition
   - I– node system : Identify file
   - Generation number
   - File system id is stored in super block
   - Generation number is stored in I- node
3. **Client**
   - It provides transparent interface to NFS.
   - Mapping between remote file name and remote file address is done at server boot time through remote mount.

2.8 Introduction to Name Services

A name service stores a collection of one or more naming contexts - sets of bindings between textual names and attributes for objects such as users, computers, services and remote objects. The major operation that a name service supports is to resolve a name. that is, to look up attributes from a given name- that is, name services store the attributes of objects in a distributed system in particular, their addresses and return these attributes when a textual name is supplied to be looked up. The main requirements for the name service are an ability to handle an arbitrary number of names, a long lifetime, high availability, the isolation of faults, and the tolerance of mistrust.

2.9 Domain Name Services (DNS)

DNS stands for Domain Name System. The Domain Name System is a name service design whose main naming database is

used across the Internet. DNS replaced the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them.

The main objective of DNS is to translate domain to IP address and vice-versa. Even though there are domain name for all websites, there are also IP address for them. Internet uses the these IP address to identify the websites. IP address is a numerical data incorporated with 4 parts separated by dots(-). This numerical value isn't easy to remember so domain name are created which are easily memorable. DNS is responsible for translating these domain names to IP addresses.

Domain names: The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called generic domains) in use across the Internet were

com - Commercial organizations

edu- Universities and other educational institutions

gov- US governmental agencies

net- Major network support centres

org-Organizations not mentioned above

int-International organizations

In addition, every country has its own domains:

us-United States

uk- United Kingdom

fr-France

np-Nepal

DNS queries : The Internet DNS is primarily used for simple host name resolution and for looking up electronic mail hosts, as follows:

Host name resolution: In general, applications use the DNS to resolve host names into IP addresses.

Mail host location: Electronic mail software uses the DNS to resolve domain names into the IP addresses of mail hosts – i.e., computers that will accept mail for those domains.

DNS is the internet naming scheme that allows resources to be accessed by using alphanumeric name. Some of the important of naming are:

1. Names are used to share resources, uniquely identity entities, to refer locations and so on.
2. Name resolution allows a process to access the named entity.

   A valid DNS allows

   i. A to Z

   ii. a to z

   iii. 0 to 9

   iv. Hyphen (-)

DNS is a distributed hierarchical database

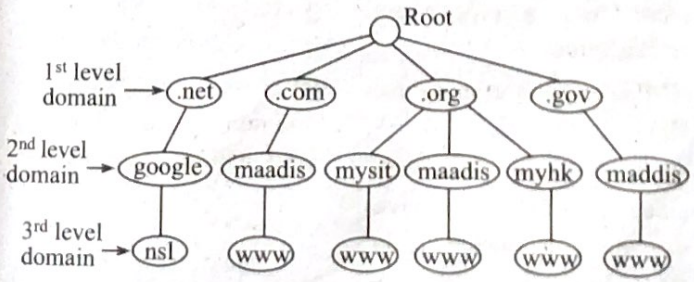


Fig: DNS

DNS Working Mechanism

DNS translates the domain name into IP addres automatically. Following steps will take you through the step included in domain resolution process:

1. Issue a DNS query to ask for IP address of www.example.com.
2. Issue a query to root name server.
3. Returns the IP address of TLD (Top Level Domain)
4. Issue a query and send to TLD servers
5. Reply with ns1.example.com and IP Address.
6. Issue another query and send to ns1.example.com
7. Reply IP address of www.example.com
8. Return IP address to client
9. Request for web content to the designated IP Address

Differences between RMI and RPC:

RMI	RPC
1. RMI is limited to Java	▪ RPC is language neutral
2. RMI is object oriented	▪ RPC is procedure oriente like–c'
3. RMI allows objects to be passed as argument and return values	▪ RPC supports only primitiv data type
4. RMI is easy to program than RPC	▪ RPC is a bit difficult t program when compared t
5. RMI is slower than RPC since RMI involves execution of Java byte code	RMI. RPC is faster than RMI
6. RMI allows usage of design patterns due to the object oriented nature	▪ RPC doesn't have th capability to use desig patterns

CHAPTER-3

OPERATION SYSTEM SUPPORT

CHAPTER-3

OPERATION SYSTEM SUPPORT

3.1 Distributed Operating System

Distributed OS is an OS that procedures a single system image for all the resources in the distributed system. It provides essential services and functionality of an OS as well as adds attributes and configurations that allows support for additional requirements like scaling and availability.

3.2 Network Operating System

Network operating system is an operating system that allows multiple resources to communicate share files and hardware devices with each other example: UNIX, windows and Mac OS.

Differences between distributed OS and network OS:

Basis for comparison	Network OS	Distributed OS
1. Objective	Provision of local services to the remote client.	Management of hardware resources
2. Architecture	2 tier client/ server architecture	N–tier client/ server architecture
3. Level of transparency	Low	High
4. Basis for communication	Files	Shared memory and messages
5. Resources management	Handled at each node	Global central or distributed management
6. Ease of Implementation	High	Low
7. Scalability	More	Less or moderate
8. Openness	Open	Closed
9. Rate of autonomy	High	Low
10. Fault tolerance	Less	High

3.3 Operating System Layer

The middle ware and OS combination should provide good performance. The operating system running at a node is known as kernel. The kernel and associated user level services provides abstractions of local hardware resources for processing, storage and communication.

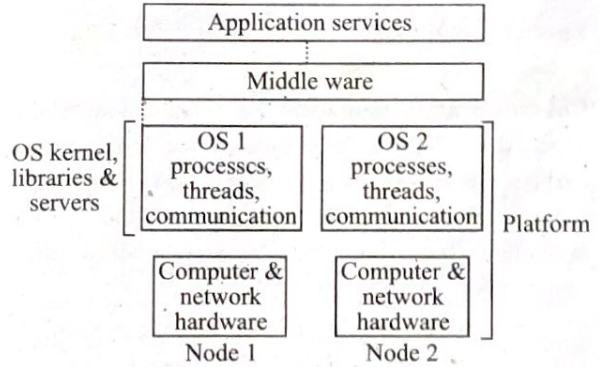


Fig: System layer

Above figure shows how the operating system layer at each of two nodes supports a common middle ware layer in providing a distributed infrastructure for application and services.

Kernel and server processes are the components that manages resources and presents clients with an interface to the resources.

The OS facilitates:

i. Encapsulation
ii. Protection
iii. Concurrent processing

Invocation mechanism is the means of accessing an encapsulated resources.

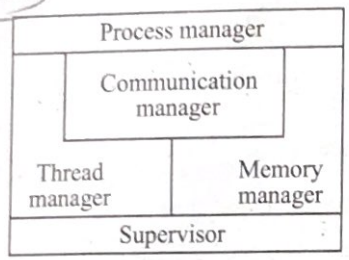


Fig: Core OS functionality

1. **Process manager:** Creation of and operations upon processes. A process is a unit of resource management, including an address space and one or more threads.
2. **Thread manager:** Responsible for thread creation, synchronization and scheduling
3. **Communication manager:** Communication between threads attached to different processes on the same computer. Some kernels also support communication between threads in remote processes. Other kernels have no notion of other computers built into them, and an additional service is required for external communication.
4. **Memory manager:** Deals with management of physical and virtual memory to ensure efficient data sharing
5. **Supervisor:** Responsible to dispatch interrupts, system call traps, memory management control, hardware cache and so on

3.4 Protection

The underlying resources in a distributed system should be protection from illegitimate access. Protection of resources should be handled by operating system layer.

Example: Consider a file with read access for client A, the illegimate access would be write access to the file be client A.

3.5 Kernel

Kernel is a program that is executed with complete access privileges for the physical resources on its host computer. control memory management and ensures access to physical resources by acceptable processes only. A kernel process executes with the supervisor(privileged) mode;the kernel arranges that other processes execute user(unprivileged) mode . It setup address space for all the processes. A process is unable to access memory beyond its address space. A program is able to switch the address space via interrupt or system call trap. There are 2 type of kernel.

i. Monolithic kernel
ii. Microkernel

Monolithic	Microkernel
1. Kernel size is large	Kernel size is small
2. OS is complex to design	OS is easy to design, implement and install.
3. Request may be serviced faster	Request may be serviced slower than monolithic.
4. All operating system services are included in kernel	Kernel only IPC and low level device management services.
5. No message passing and no context switching is required while kernel is performing job.	It requires message passing and context switching i
6. E.g., UNIX and LINUX kernel.	E.g,, Kernel of MAC OS, windows NT.

3.6 Process and Threads

Process

A process consists of an execution environment together with one or more threads. A thread is the operating system abstraction of an activity. An execution event is the unit of resources

management; a collection of local kernel managed resources to which its threads have access.

An execution environment consists of:

- An address space.
- Thread synchronization and communication resources. (example: semaphores, sockets)
- Higher-level resources. (example.: file system, windows)

Threads

Threads are schedule activities attached to process. The aim of having multiple threads of execution is:

- To maximize degree of concurrent execution betwee operations.
- To enable the overlap of computation with input an output.
- To enable concurrent processing multiprocessors thread can be helpful within servers:
- Concurrent processing of clients requests can reduce th tendency for servers to become bottleneck.

Example: One thread can process a client's request while second thread serving another request wait's for disk access t complete.

Differences between process and thread:

Process	Thread
1. A process is a program under execution i.e. an active program	A thread is a lightweight proces that can be manage independently by a scheduler
2. A processes require more time for context switching as they are more heavy.	A threads require less time fo context switching as they ar lighter than processes.
3. Processes are totally independent and don't share memory.	A thread may share memory with its peer threads.

Process	Thread
4. Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes.
5. If a process get blocked, remaining processes can continue execution.	If a user level threads gets blocked, all of its peer threads also get blocked.
6. Individual process are individual to each other	Threads are part of a processes and so are dependent.
7. Processes require more time for terminate.	Thread require less time for termination.
8. Processes require more resources than threads	Threads generally need less resources than processes
9. Processes have independent data & code segments	A thread shares the data segment, code segments, files etc. with its peer threads
10. Individual processes are independent of each other	Threads are parts of a process & so dare dependent.

3.7 Communication and Invocation

Here, we concentrate on communication as part of the implementation of what we have called an invocation a construct, such as remote method invocation, remote procedure call or event notification, whose papoose is to bring about an operation on a resources in a different address space.

We shall cover operating system design issues and concepts by asking the following questions about the OS:

- What communication primitives does it supply?
- Which protocols does it support & how open is the communication implementation?
- What steps are taken to make communication as efficient as possible?

- Communication primitives: do operation, get request and sen reply.
- OS communication: what, which protocols.
- Protocols and openness: most OS in 1980s incorporated the own network protocols tuned to RPC interactions.

Invocation performance

- Client and server may take many millions of invocatio related operations in their lifetimes.
- However the network bandwidth improves invocation time have not decreased in proportion.
- NULL RPC is defined as an RPC without parameter tha executes a null procedure and return no values.
  - It's execution involves an exchange of message carryin some system data but not user data.
  - Null invocation costs are important because they measur a fixed overhead, the latency.
  - Invocation cost increase with the size of arguments an results, but in many cases the latency is significan compared with the remainder of delay.
  - RPC bandwidth/ throughput is also concern when dat has to be transferred in bulk.
- Marshalling and unmarshaling: It involve copying an converting data become significant when amount of dat grows.
- Data copying: message data is copied several times in th course of RPC like from across s the user kernel boundary, cross each protocol layer, between network interface an kernel buffers.
- Packets initialization: initializing protocol headers and transfe including checksums.
- Thread scheduling and context switching:

Twomajor kernel architecture:

- Monolithic kernels
- Microkernel

Hybrid Approaches

- Pure microkernel operating system such as chorus & mach have changed over a time to allow servers to be loaded dynamically into the kernel address space or into a user level address space.
- In some operating system such as SPIN, the kernel and all dynamically loaded modules grafted onto the kernel execute within a single address space.
- An open distributed system should make it possible to
  - Run only that system software at each computer that is necessary for its particular role in the system architecture.
  - Allow the software (and the computer) implementing any particular service to be changed independent of other facilities.
  - Allow for alternatives of some service to be provided, when this is required to suit different users or applications.
- A guiding principle of operating system design
  - The separation of fixed resource management "Mechanism" from resource management "policies", which vary from application to application and service to service.
  - The kernel would provide only the most basic mechanism upon which the general resource management tasks at a node are carried out.
  - Server modules would be dynamically loaded as required, to implement the required resource management policies, for the currently running applications.

CHAPTER-4

DISTRIBUTED HETEROGENEOUS APPLICATION AND CORBA

CHAPTER-4

DISTRIBUTED HETEROGENEOUS APPLICATION AND CORBA

4.1 Heterogeneity in Distributed System

Distributed application are typically heterogeneous.

- Different hardware: mainframes, workstations, personal computers, Real Time Operating System etc.
- Different software: UNIX, windows, IBM OS/2, Real time Operating System etc.
- Unconventional device: Telephone switch, teller machines.
- Diverse network: Ethernet, ATM

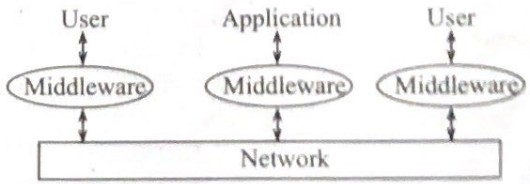
Why heterogeneity in distributed system?

- Different hardware/software solutions are considered to be optimal for different part of system.
- Different users have to interact are deciding for different hardware/software solutions/vendors.

4.2 Middleware

Middleware is the key component of a heterogeneous distributed client- server environment.

Middleware is a set of services that enable applications and end users to interact each other across heterogeneous distributed system. Middleware software resides above the network and below the applications software.



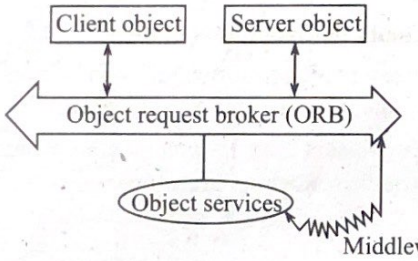
Middleware should make the network transparent to the applications and end users ⇒ users and applications should be able to perform the same operations across the network that they can perform locally. Middleware should hide the details of computing hardware, OS, software components, across networks.

Different kinds of software qualities, to certain extent as middleware:

- FTP and Email
- Web browsers
- CORBA

4.3 Objects in Distributed System

A distributed application can be viewed as collection of objects (user interfaces, databases, application module, customers).



Object: Objects are data surrounded by code; each one has its own attributes and methods which define the behavior of the object; objects can be clients, server or both.

Object request broker (ORB)

- It allows objects to find each other and interacts between them over the network.

Object services

- It allows to create, name, move, copy, store, delete, restore and manage objects.

4.4 Interface Definition Language (IDL)

An interface specifies the API (Application Programming Interface) that the clients can use to invoke operations on objects:

- the set of operations
- the parameters needed to perform the operations.

An interface specifies the API used by the clients to invoke operations on the objects.

- Each objects can have one or more interfaces.
- Interfaces are defined with the help of IDL.
- Middleware supports parsing of IDL of an interface with help of interface compilers.
- IDL only specifies declarative, not the code.
- IDL should be independent of implementation language.
- IDL defines capacity of the distributed services along with common set of data types of inter acting with those services.

4.5 The CORBA Approach and Services

Common request broker architecture (CORBA) is the standard developed by the object management group to provide interoperability among distributed objects.

It is the world's leading middleware solution enabling the.

i. Exchange of information
ii. Independent of hardware platform
iii. Programming language
iv. Operating system

CORBA is essentially designed specification for an object request broker where, ORB provides the mechanism required for distributed object to communicate with me another CORBA is software based communication.

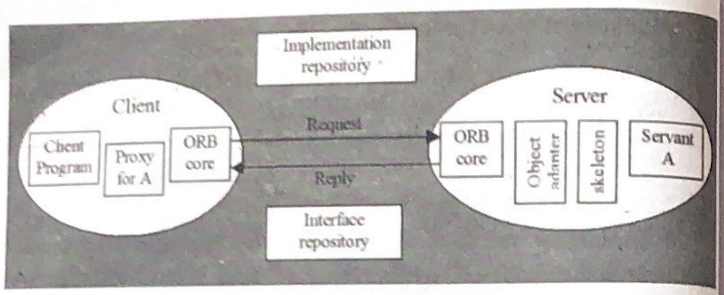


Fig: Architecture of CORBA

ORB core:

The role of the ORB core is similar to that of the communication module. In addition an ORB core provides an interface that includes the following:

- operations enabling it to be started and stopped
- operations to convert between remote object references and strings;
- operations to provide argument lists for requests using dynamic in vocation .

Object adapter (server):

The role of an object adapter is to bridge the gap between COBRA objects with IDL interfaces and the programming language interface of the corresponding servant classes. This role also include that of the remote reference and dispatcher modules. An object adapter has the following tasks:

- It creates remote object references for COBRA objects.
- It dispatches each RMI via a skeleton to the appropriate servent.
- It activates and deactivates servants.

Skeletons (server) :

Skeleton classes are generated in the language of the server by an IDL compiler. As before, remote method invocations are di spatched via the appropriate skeleton to a particular servant, and the skeleton unmarshals the arguments in request messages and marshals exceptions and result s in reply messages.

Client stubs/proxies:

These are in the client language. The class of a proxy (for object-oriented languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language. As before, the client stubs/ proxies marshal the arguments in invocation requests and unmarshal exceptions and results in replies.

Implementation repository

An implementation repository is responsible for activating registered servers on demand and for locating servers that are currently running. The object adapter name is used to refer to servers when registering and activating them.

Interface repository

The role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it. For an interface it can supply the names of the methods and for each method, the names and types of the arguments and exceptions. Thus, the interface repository adds a facility for reflection to CORBA. Suppose that a client program receives a remote reference to a new CORBA object. Also suppose that the client has no proxy for it ; then it can ask the interface repository about the methods of the object and the types of parameter each of them requires.

CORBA services and description:

CORBA includes specifications for services that may be required by distributed objects. The CORBA services include the following:

Naming and Trading Services:

- The basic way an object reference is generated is at creation of the object when the reference is returned.
- Object references can be stored together with associated information (e.g. names and properties).

- Object references can be stored together with associated information (e.g. names and properties).
- The trading service allows clients to find objects based on their properties.

Transaction Management Service:

- It provides two phase commit coordination among recoverable components using transactions.

Concurrency Control Service:

- It provides a lock manager that can obtain and free locks for transactions or threads.

Security Service:

- It protects components from unauthorized users; it provides authentication, access control lists, confidentiality, etc.

Time Service:

- It provides interfaces for synchronizing time; provides operations for defining and managing time-triggered events.

Life Cycle Service:

- The life cycle service defines conventions for creating, deleting copying and moving CORBA objects. It specifies how clients call use factories to create objects in particular locations, allowing persistent storage to be used if required. It defines an interface that allows clients to delete CORBA objects or to move or copy them to a specified location.

CHAPTER-5

TIME AND STATE IN DISTRIBUTED SYSTEM

CHAPTER-5

TIME AND STATE IN DISTRIBUTED SYSTEM

5.1 Time in Distributed System

Time is an important and interesting issue in distributed systems, for several reasons. First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time. For example, an 'e-Commerce' transaction involves events at a merchant's computer and at a bank's computer. It is important, for auditing purposes, that those events are timestamped accurately.

Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution. These include maintaining the consistency of distributed data , checking the authenticity of a request sent to a server and eliminating the processing of duplicate updates.

5.1.1 Physical Clock

Each node consists of a physical device that counts oscillation in a crystal at a definite frequency and store division of count to frequency in a register to provide time. Such device is called physical clock and the time calculated is called physical time.

There are three methods of synchronization of physical clock:

1. Cristian's method
2. Berkeley's method
3. Network time protocol (NTP)

1. Cristian's Method

Cristian [1989] suggested the use of a time server, connected to a device that receives signals from a source of universal time coordinated (UTC), to synchronize computers externally.

Upon request, the server process 'S' supplies the time according to its clock, as shown in Figure below . Cristian observed that while there is no upper bound on message transmission delays in an asynchronous system, the round-trip times for messages exchanged between pairs of processes are often reasonably short – a small fraction of a second. He describes the algorithm as probabilistic: the method achieves synchronization only if the observed round-trip times between client and server are sufficiently short compared with the required accuracy.

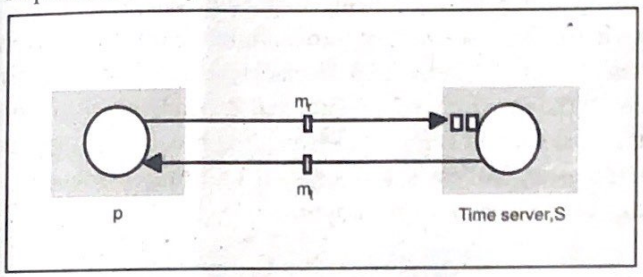


Fig.: Clock synchronization using s time server.

Algorithm:

1. A process 'P' requests the time in a message 'm_r' and receives the time value 't' in a message 'm_t'.
2. 't' is inserted in 'm_t' at the last possible point before transmission from nodes 'S'.
3. Process 'p' records the total round trip time. 'T_{round}' taken to send the request 'm_r' and receives the reply 'm_t'.
4. M_{in}= minimum queuing time for S.
5. Clock is set by: $t+T_{round}/2$.

Drawbacks:

- If the time server fails, the synchronization is impossible
- This drawbacks can be removed by providing a group of synchronized time server.

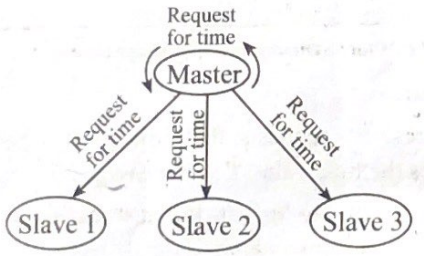
Needs of Cristian's method:

The processes can run on different machines and no global clock to judge which event happens first.

2. Berkeley's Algorithm

Berkeley algorithm uses internal synchronization.

Unlike in Cristian's protocol, this computer periodically polls the other computers whose clocks are to be synchronized called slaves. The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times (similarly to Cristian's technique), and it averages the values obtained (including its own clock's reading). The balance of probabilities is that this average cancels out the individual clocks' tendencies to run fast or slow. The accuracy of the protocol depends upon a nominal maximum round-trip time between the master and the slaves. The master eliminates any occasional readings associated with larger times than this maximum.



Algorithm:

1. A master is chosen via an election process.
2. The master polls the slaves who reply with their time in a similar way to Christian's algorithm.
3. The masters observes the round trip time (RTT) of the messages and estimate the time of each slave & it's own.
4. The master then averages the clock times, ignoring any values it receives for outsides the value of the others.

5. Instead of sending the updated current time back to the process, the master then spends out the amount (positive or negative) that each slave must adjust its clock.

   This avoids further uncertainty due to RTT at the slave process.

Example:

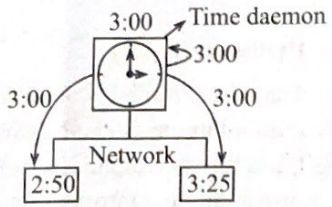


Fig. (a): The time daemon asks all the other machines for their clock values.

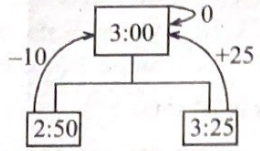


Fig. (b): The machines answer

$$\text{avg} = \frac{0 - 10 + 25}{3} = \frac{15}{3} = +5$$

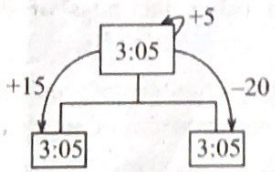


Fig. (c): The time daemon tells everyone how to clock adjust their clocks.

Drawbacks:

i. Time not a reliable method of synchronization.

ii. Users mess up clock.

(and forget to set their time zones).

iii. Unpredictable delay in internet.

iv. Relativistic issues

If A and B are for apart physically and two event T_A and T_B are very close in time, then which comes first? how do you know?

3. Network Time Protocol (NTP)

Cristian's method and the Berkeley algorithm are intended primarily for use within intranets. The Network Time Protocol (NTP) [Mills 1995] defines an architecture for a time service and a protocol to distribute time information over the Internet.

NTP can be defines an architecture to enable client across the internet to be synchronized accurately to universal time coordinated (UTC). It synchronizes against many time servers.

The design aims of NTP are:

i. To provide a service enabling clients across the Internet to be synchronized accurately to UTC.

ii. To provide a reliable service that can survive lengthy losses of connectivity.

iii. To enable clients to re-synchronize sufficiently frequently to offset the rates of drift found in most computers.

iv. To provide protection against interference with the time service, whether malicious or accidental:

Hierarchical structure of NTP

i. NTP service is provided by a network of server located across the internet.

ii. Primary servers are directly connected to a time server.

iii. Secondary servers are synchronized with primary servers.

iv. The logical hierarchy of the server connected is known as synchronization subnet.

v. Each level of synchronization subnet is known as stratum.

vi. The lowest level executes in user's work stations.

vii. Server with high stratum numbers are viable to have less accurate clocks.

Server synchronization in NTP

- Multicast mode
  - Server periodically multicasts time to other servers in the network.
  - Receives set their clock assuming small delays.
- Procedure call mode
  - One server accepts request from other nodes.
  - Server replies with its time stamp.
- Symmetric mode
  - A pair of servers an higher subnet layers exchange messages to improve accuracy of synchronization over time.

Clock synchronization in NTP

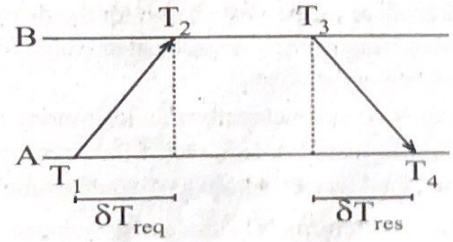


Fig. : Getting the current time from a time server.

In this case, A will send a request to B, timestamped with value T_1.B, in turn, will record the time of receipt T_2 (taken from

its own local clock), and returns a response timestamped with value T_3, and piggybacking the previously recorded value T_2. Finally, A records the time of the response's arrival, T_4. Let us assume that the propagation delays from A to B is roughly the same as B to A, meaning that $T_2 - T_1 \approx T_4 - T_3$. In that case, A can estimate its offset relative to B as

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

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

Of course, time is not allowed to run backward. If A's clock is fast, $\theta < 0$, meaning that A should. In principle, set its clock backward. This is not allowed as it could cause serious problems such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change

In, the case of the network time protocol (NTP), this protocol is set up pair wise between servers. In other words, B will also probe A for its current time. The offset θ is computed as given above, along with the estimation δ for the delay.

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

Eight pair of (θ, δ) values are buffered, finally taking minimal value found for δ, as the best estimation for the delay between two servers, and subsequently the associated value θ as the most reliable estimated of the offset.

Applying NTP symmetrically should in principle, also let B adjust it's clock to that of A. however if Bk's clock is known to be more accurate, then such an adjustment would be foolish.

To solve this problem, NTP divides servers into strata.

5.1.2 Logical Clock

Logical clock is the virtual clock that records the relative ordering of events in a process. It can be implemented using a monotonically increasing software counter.

1. Lamport's Clock Algorithm

To synchronize logical clocks, Lamport defined a relation called **happens-be-fore**. The expression $a \rightarrow b$ is read "a happens before b" and means that all processes agree that first event a occurs, then afterward, event b occurs. The happens-before relation can be observed directly in two situations:

1. If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$ is true.
2. If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then $a \rightarrow b$ is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite, nonzero amount of time to arrive.

Each process P_i keeps its own logical clock L_I which is used to apply lamport timestamp to events.

The ordering of events depends on two situations.

1. If two events within a same process occurred, they occurred in the order in which that process observes.
2. When ever a message is sent between processes the event of sending message occurred before the event of receiving message.

Drawbacks:

Distinct event of different processes can have same timestamp (partial ordering of events).

Implementation rule L_i logical clock of process P_i.

1. L_i is incremented before each event is issued at process p_i.

   $L_i = L_i + 1$

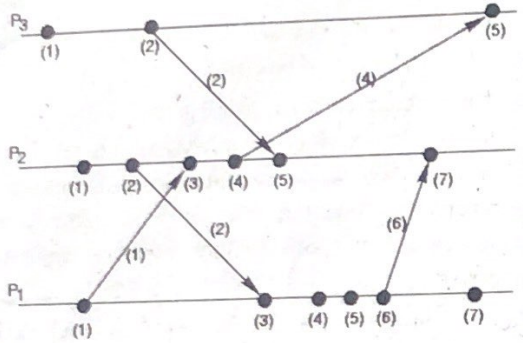
2. When send (m) is a event of P_i,

   Timestamp $t_M = L_i$ is included in message m.

3. On receive (m) by P_j ,its clock L_j is updated $\{L_j = \max [L_j, t_M]\}$. The new value of L_j is used to timestamp event receive(m) by p_j.

Example:

Lamport Logical Clock (Example)



2. **Vector Clock**

Vector clock is a clock that gives ability to decide whether two events are causally related or not. A vector clock for a system of n processes is an array of N integer. each process keeps its own vector clock V_i used to time stamp local events.

Drawbacks: It takes of more storage and message payload.

Implementation rule:

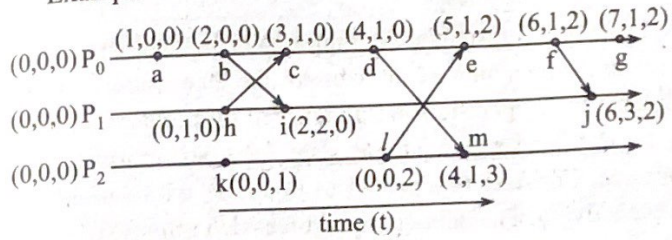
1. Initially, $V_i[j] = 0$ for, i, j= 1 to N
2. Just before P_i timestamps an event

   It sets $V_i [i] = V_i[i] +1$
3. P_i includes value $t = V_i$ in every message it sends
4. When p_i receive a timestamp t in a message, it sets

   $V_i [j] = \text{Max } [V_i[j], t[j]]$ for j = 1 to N. This operation is known as merge operation.

   For a vector clock V_i ,$V_i[i]$ is a number of events that p_i has time stamped and $v_i[j]$ is number of events that occurred at p_j, that has potentially affect p_i.

Example



5.2 Global State and State Recording

Global State

It is desirable to determine whether a particular property holds true for a distributed system as it executes. a global state predicate is a function that maps from the set of global state of processes in the system {true, false}. It is the state of the complete distributed system showing all the events occurred in different nodes of system and the relative timing order until a particular instance of item.

Types of cuts:

- A cut is said to be consistent, if for each event it contains, also contains all the events that happened before that event.
- A cut is said to be in consistent, if it contains some events that happened after an event that is not included in the cut.

In a cut,

for $a \rightarrow b$

either only a or both a and b is present the cut is consistent.

but, if only b is present the cut is inconsistent.

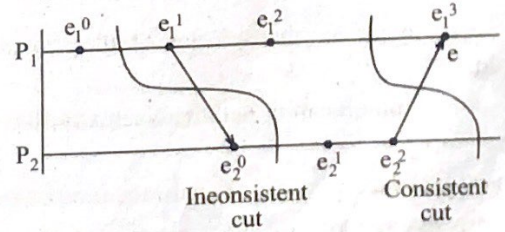
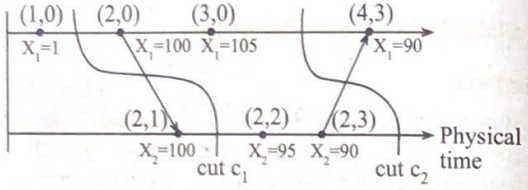


Fig: types of cut

5.3 Distributed Debugging

Distributed system are complex to debug and case needs to be taken in establishing what caused the execution. Each of the problems have specific solution tailored to it, but they all illustrate the need to observe a global state and so, motivate a general approach. Consider two processes P_1 and P_2 with variables X_1 and x_2 respectively. The debugging involves determining $|X_1 - X_2| > d$.



5.4 Distributed Snapshot

Algorithm that is used to create a consistent snapshot of the global state of a distributed system.

Chandy-Lamport Algorithm

- Snapshot algorithm that is used in distributed system for recording a consistent global state of an asynchronous system.
- The algorithm works using marker messages.
- Each process who want to initiated a snapshot records its local state and send a marker on each of the outgoing channels.

Assumptions:

- There are no failure and all message arrive intact and only once.
- The communication channel are unidirectional and FIFO ordered.
- There is a communication path between any two processes in the system.
- It doesn't interface with the normal execution of the processes.
- Each process records its local states and states of another incoming state.

Algorithm:

1. The process that initiates, the snapshot
   - saves its own local state.
   - sends a snapshot request message bearing a snapshot taken to all other processes.
2. A process receiving snapshot taken for the first time on any message.
   - sends the observer its own saved state.
   - attach a snapshot taken to all subsequent message.
3. When a process that has already received the snapshot taken receives a messages that does not bear the snapshot taken, the message is forward to the observer process.

CHAPTER-6

COORDINATION AND AGREEMENT

CHAPTER-6

COORDINATION AND AGREEMENT

6.1 Mutual Exclusion in Distributed system

Mutual exclusion is the mechanism that prevent interface and ensures consistency when accessing the resources. It ensures that concurrent processes make a serialized access to shared resources or data. It is similar to critical section problem of operating system.

In a distributed system neither shared variables (semaphores) nor a local kernel can be used in order to implement mutual exclusion! Thus, mutual exclusion has to be based exclusively on message passing, in the context of unpredictable message delays and no complete knowledge of the state of the system.

Sometimes the resource is managed by a server which implements its own lock together with the mechanisms needed to synchronize access to the resources mutual exclusion and the related synchronization are transparent for the process accessing the resource. This is typically the case for database systems with transaction processing.

Essential requirements for mutual exclusion:

The primary objective of a mutual exclusion algorithm is to maintain exclusion i.e., to guarantee that only one request accesses the CS at a time. In addition the following characteristic are considered importance in a mutual exclusion algorithm.

1. Freedom from Deadlocks
2. Freedom from Starvation.
3. Fairness
4. Fault Tolerance

1. **Freedom from Deadlocks:** Two or more sites should not endlessly wait for messages that will never arrive.

2. **Freedom from Starvation:** A site should not be forced to wait independently to execute CS while other sites are repeatedly executing CS.
3. **Fairness:** Fairness dictates that requests must be executed in the order they are made. Since a physical global clock does not exist, time is determined by logical clocks. Note that fairness implies freedom from starvation, but not vice-versa.
4. **Fault Tolerance:** A mutual exclusion algorithm is fault-tolerant if in the wake of a failure, it can reorganize itself so that continues to function without any disruptions.

Assumption for mutual exclusion:

1. The system should be asynchronous
2. The processes don't fail
3. The message delivery is reliable

Protocol for mutual exclusion:

The application level protocol for executing a critical section is as follows:

enter ( ),// enter critical section -block if necessary.

resource Access ( ),//access shared resources in critical section

exit ( ).//leave critical section-other processes may now enter.

Performance criteria for mutual exclusion:

1. **Bandwidth consumed**

   Number of message sent in each entry and exit operation.

2. **Client delay**

   Incurred by a process at each entry and exit operation.

3. **Throughput**

   Rate at which the collection of processes as a whole can access the critical section assuming some communication is necessary between successive processes.

There are two basic approaches to distributed mutual exclusion:

1. Non Token based algorithm
2. Token-based algorithm

6.2.1 Non Token Based Algorithm

Each process completes to gain access to use the shared resources; Request are arbitrated by central control site or by distributed agreement.

The non taken based algorithms are:

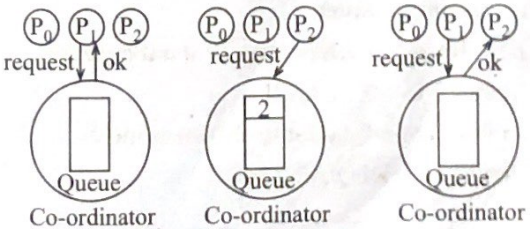
1. Central coordinator algorithm
2. Ricart-Agrawala algorithm

1. Central Coordinator Algorithm

- A central coordinator is present within the distributed system which is responsible to grant access to enter a critical section.
- A request queue is maintained by the central coordinator to handle multiple requests efficiently and effectively.

Algorithm

1. To enter a CS a process sends a request message to coordinator work while waiting for a reply (during this waiting period the process can continue with other work).
2. The reply from coordinator gives right to access critical section based on request queue.
3. After finishing critical section operation, the process notifies coordinator with a release messages.



Advantages:

1. Easy to implement
2. Require only 3 message per access to critical section.

Disadvantages:

1. The coordinator can become a performance bottleneck.
2. The coordinator is a critical point of failure:

- If the coordinator crashes, a new coordinator must be created.
- The coordinator can be one of the processes competing for access; an election algorithm has to be run in order to choose one and only one new coordinator.

2. Ricart-Agrawala Algorithm

It uses distributed agreement to implement mutual inclusion. All process are assumed to keep Lamport's logical clock. The process requiring access to critical section multicast request message to all other process competing for same resource.

A process enters critical section when all processes reply to the message.

Each process keep its state with respect to critical section. The possible states are:

- Released
- Requested
- Held

Algorithm:

1. Rule for Initialization

   /* performed by each process P_i at initialization */

   - State (P_i) = RELEASED.

2. Rule for Access Request to critical section

   - State (P_i) = REQUESTED
   - $T(P_i)$ = the value of the local logical clock corresponding to this request.
   - P_i sends request message to all processes; the message is of the form $[T(p_i),i]$, where i is an identifier of P_i.
   - P_i waits until it has received replies from all n–1 processes

3. Rule for executing critical section

   /* performed by P_i after it received the n-1 replies */

   - State (P_i) = HELD.

   P_i enters the CS.

4. Rule for handling incoming requests.

   /* performed by P_i whenever it received a request ($T(p_j)$, j) from P_j*/

   - If state (P_i) = HELD or [state(P_i) = REQUESTED and $[\{T(P_i),i)\} < \{T(P_j), j\}]$, then put request to queue without replying.
   - ELSE, Reply immediately to p_j.

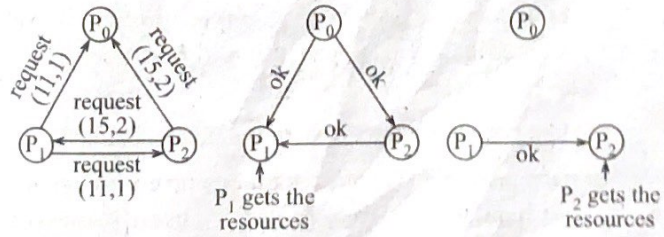
   End if.

5. Rule for RELEASING CRITICAL SECTION.

   /* performed by P_i after it finished work in a CS */

   - state (P_i) = RELEASED
   - P_i replies to all queued requests.

A request issued by a process P_j is blocked by another process P_i only if P_i is holding the resource or if it is requesting the resource with a higher priority (this means a smaller timestamp) then P_j.



Problems:

1. The algorithm is expensive in terms of message traffic; it requires 2(n–1) requests and (n–1) messages for entering a CS:(n-1) requests and (n-1) replies.
2. The failure of any process involved makes progress impossible if no special recovery measures are taken.

6.2.2 Token Based Algorithm

A logical token is passed among the processes. A token represents access right to the shared resources. The process which holds the token is granted access to the critical section.

1. Ricart-Agrawala second algorithm
2. Token Ring algorithm

1. Ricart-Agrawala Second Algorithm

A process is allowed to enter the critical section when it got the token. In order to get the token it sends a request to all other processes competing for the same resource. The request message consists of the requesting process' timestamp (logical clock) and its identifier.

Initially the token is assigned arbitrarily to one of the processes.

- When a process P_i leaves a critical section it passes the token to one of the processes which are waiting for it; this will be the first process P_j , where j is searched in order [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] for which there is a pending request.

  If no process is waiting, P_i retains the token (and is allowed to enter the CS if it needs); it will pass over the token as result of an incoming request.

- How does P_i find out if there is a pending request?

  Each process P_i records the timestamp corresponding to the last request it got from process P_j, in request(p_i[j]). In the

token itself, token[j] records the timestamp (logical clock) of P_j's last holding of the token. If request P_i[j] > token[j] then P_j has a pending request.

Algorithm:

1. Rule for initialization

/* performed at initialization */

- state(p_i) = NO-TOKEN for all processes P_i except for one single process p_x for which (state(p_x)) = TOKEN PRESENT.
- token [K] = 0 for all k = 1 to n.
- $request_{pi}$ [K] = 0 for all p_i and k=1 to n.

2. Rules for access request and critical section execution

/* performed whenever process P_i requests an access to the CS and when it finally gets it; in particular P_i can already possess the token */

i. State (p_i) = NO-TOKEN :

p_i sends request message to all processes; the message is on the form [T (p_i ,i)] where $T_{Pi} = C_{Pi}$ is the value of the local logical clock, and i is an identifier of Pi.

p_i waits until it receives token.

ii. State (p_i) = TOKEN-HELD

p_i enters in critical section.

3. Rules for handling incoming request

/* performed by P_i whenever it received a request (T_{Pj}, j) from P_j */

i. $Request_{pi}$[j] = max [$request_{pi}$[j], T (p_i)]

ii. If state (p_i) = TOKEN-PRESENT

p_i releases the token

End if.

4. Rules for releasing critical section

/* performed by P_i after it finished work in a CS or when it holds a token without using it and it got a request */

i. State (p_i) = TOKEN-PRESENT

ii. for k = [i+1,i+2.....n, 1,2.....:i–2,i–1]

do:

if $request_{pi}[k]$ >token [k] :then

State(p_i) = NO-TOKEN.

token [i] = C(p_i)

p_i sends token to p_K .

break.

End if.

End for.

5. Each process keeps its state with respect to the token: NO-TOKEN, TOKEN-PRESENT, TOKEN-HOLD.

State diagram

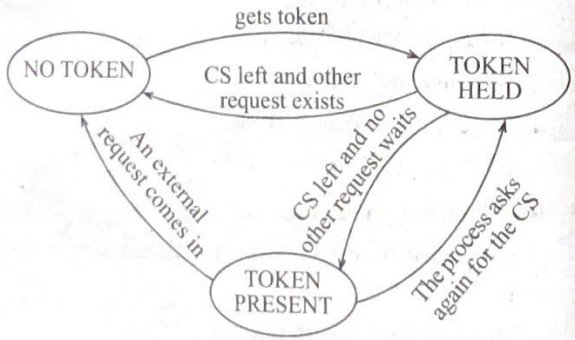
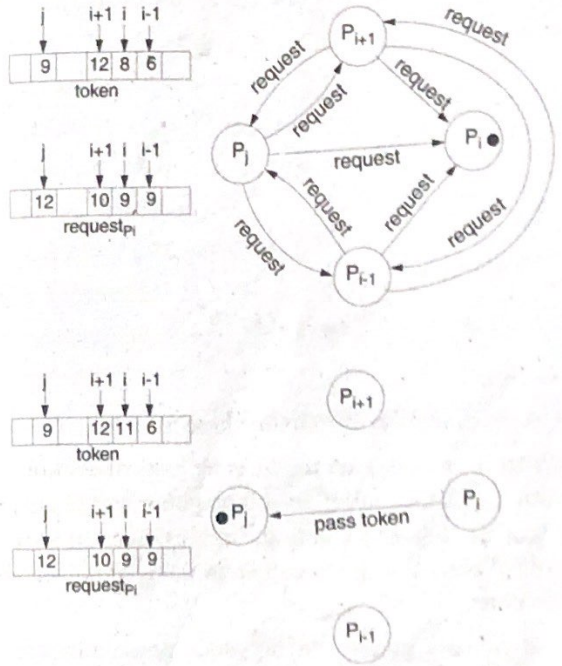


Fig.: State diagram of Ricart- Agrawala second algorithm

Example:



Advantages:

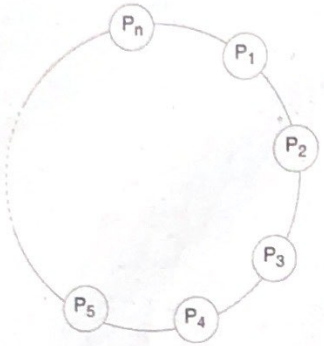
1. The complexity is reduced compared to the (first) Ricart-Agrawala algorithm: it requires n messages for entering a CS: (n-1) requests and one reply.
2. Failure of process which is not holding token does not prevent progress.

2. Token Ring Algorithm

A very simple way to solve mutual exclusion is to arrange the n processes P_1, P_2, ... P_n in a logical ring.

- The logical ring topology is created by giving each process the address of one other process which is its neighbor in the clockwise direction.
- The logical ring topology is unrelated to the physical interconnections between the computers.

The n processes are arranged in a logical ring.



Algorithm

1. Token is initially given to one process.
2. When a process requires to enter critical section, it waits until it gets token from its left neighbor and retains it. After it got the token it enters the critical section. After it left critical section, it passes token to its neighbor in clockwise direction.
3. If a process gets token but does not require to enter a critical section, it immediately passes token along a ring.

Problems:

1. It adds loads to the networks as token should be passed even the process does not need it.
2. If one process fails, no progress is possible until the faculty process it extracted from the ring.
3. Election process should be done if the processes holding the token fails.

6.3 Distributed Election

Election algorithm is an algorithm choosing a unique process to play a particular role. Any process can be elected but only one process must be elected and all other processes must agree on the decision.

Election is done after a failure of current coordinator or failure of process currently holding the token.

An election process is typically performed in two phases.

1. Select a leader with the highest priority
2. Inform all processes about the winner.

The distributed election algorithm are

1. Bully algorithm
2. Ring based algorithm

1. Bully Algorithm

A process has to know the identifier of all other processes (it doesn't know, however, which one is still up); the process with the highest identifier, among those which are up, is selected.

Any process could fail during the election procedure.

When a process P_i detects a failure and a coordinator has to be elected, it sends an election message to all the processes with a higher identifier and then waits for an answer message:

- If no response arrives within a time limit, P_i becomes the coordinator (all processes with higher identifier are down) then it broadcasts a coordinator message to all processes to let them know.
- If an answer message arrives, P_i knows that another process has to become the coordinator then, it waits in order to receive the coordinator message. If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the answer message) P_i re sends the election message.

When receiving an election message from P_i , a process P_j replays with an answer message to P_i and then starts an election procedure itself, unless it has already started one then it sends an election message to all processes with higher identifier.

Finally all processes get an answer message, except the one which become the coordinator

Algorithm

1. By default the state of a process is ELECTION- OFF
2. Rule for election process initiator

   /* performed by a process P_i, which triggers the election procedure, or which starts an election after receiving itself an election message */

   i. State (P_i) = ELECTION-ON

   ii. P_i sends election message to processes with higher identifier and waits for answer message.

   iii. If no answer message before timeout, P_i is coordinator and sends coordinator message.

   iv. ELSE, P_i waits for coordinator message and if no coordinator message arrives before timeout, restart election.

3. Rule for handling incoming election message

   /* performed by a process P_i at reception of an election message coming from P_j */

   i. P_i replies with an answer message to P_j

   ii. If state (P_i): ELECTION-OFF, start election procedure, according to 2.

Example:

In Fig. below we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely 5, 6, and 7. as shown in Fig.(a). Processes 5 and 6 both respond with OK, as shown in Fig.(b). Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

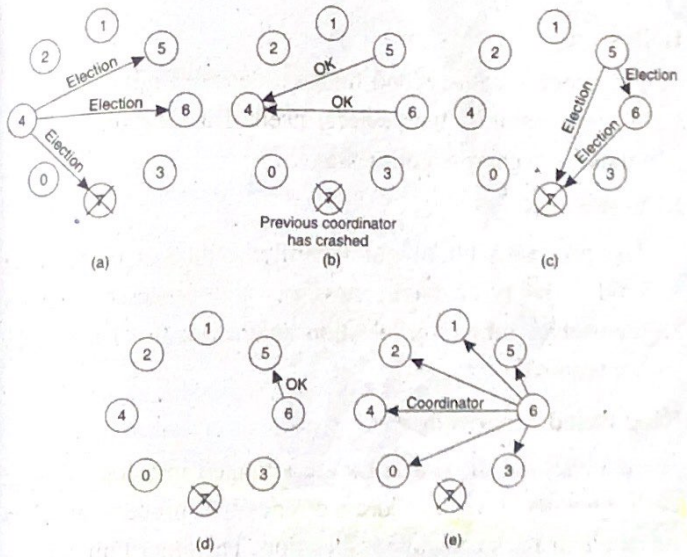


Figure: The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond. telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

In Fig(c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Fig.(d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a COORDINATOR message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue. If process 7 is ever restarted, it will just send an the others a COORDINATOR message and bully them into submission.

1. **Best case**

   The process with second highest identifier notices failure of coordinator, it then selects itself if as coordinator and sends (n-2) coordinator message.

2. **Worst case**

   The process with lowest identifier indicates election. It sends (n-1) election message to processes which themselves initiate an election again. So, $0(n^2)$ messages are required.

2. **Ring Based Algorithm**

We assume that the processes are arranged in a logical ring; each process knows the address of one other process, which is its neighbor in the clockwise direction. The algorithm elects a single coordinator, which is the process with the highest identifier.

Election is started by a process which has noticed that the current coordinator has failed. The process places its identifier in an election message that is passed to the following process. When a process receives an election message it compares the identifier in the message with its own. If the arrived identifier is greater, it forwards the received election message to its neighbor; if the arrived identifier is smaller it substitutes its own identifier in the election message before forwarding it. If the received identifier is that of the receiver itself then this will be the coordinator. The new coordinator sends an elected message through the ring.

Algorithm

1. By default, the state of a process is NON-PARTICIPANT.
2. Rule for election process initiator

   /*performed by a process P_i, which triggers the election procedure*/

   i. State (P_i) = PARTICIPANT

   ii. P_i sends election message with message.id: -i to its neighbor

2. Rule for handling incoming election message:

   /* performed by a process P_j, which receives an election message */

   i. If message.id > j, then P_j forwards election message to its neighbor and state (p_j)= PARTICIPANT

   ii. ELSE if message.id < j,

   a. If state (P_j) = NON-PARTICIPANT, P_j forwards message with message.id=j, and state(P_j)= PARTICIPANT

   iii. Else, P_j is a coordinator and sends elected message.id = j to its neighbor and state(P_j) = NON-PARTICIPANT

3. Rule for handling incoming elected message

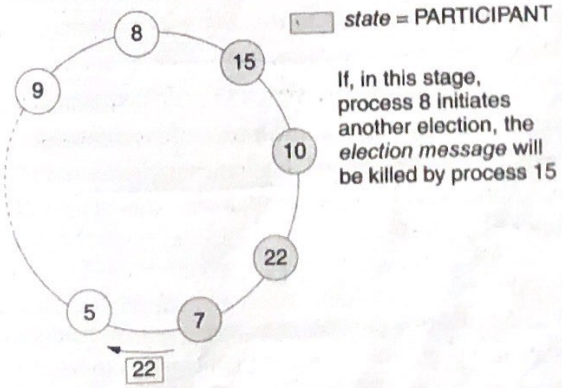
   /* performed by a process P_i, which receives an elected message */

   i. If message id ! = i :

   p_i forwards received message.

   State (P_i) = NON-PARTICIPANT

   end if.



With one single election started:

- On average: n/2 (election) messages needed to reach maximal node; n (election) messages to return to maximal node; n messages to rotate elected message.

  Number of messages: 2n + n/2.

- Worst case: n-1 messages needed to reach maximal node;

  Number of messages: 3n - 1

  The ring algorithm is more efficient on average then the bully algorithm.

6.4 Multicast Communication

Group, or multicast, communication requires coordination and agreement. The aim is for each of a group of processes to receive copies of the messages sent to the group. often with delivery guarantees. The guarantees include agreement on the set of messages that every process in the group should receive and on the delivery ordering across the group members.

The essential feature of multicast communication is that a process issues only one *multicast* operation to send a message to each of a group of processes (in Java this operation is *aSocket.send(aMessage))* instead of issuing multiple send operations to individual processes. Communication to all processes in the system, as opposed to a sub-group of them , is known as broadcast.

The use of a single *multicast* operation instead of multiple send operations amounts to much more than a convenience for the programmer. It enables the implementation to be efficient and allows it to provide stronger delivery guarantees than would otherwise be possible.

Efficiency:

The information that the same message is to be delivered to all processes in a group allows the implementation to be efficient in its utilization of bandwidth . It can take steps to send the message no

more than once over any communication link, by sending the message over a distribution tree; and it can use network hardware support for multicast where this is available. The implementation can also minimize the total time taken to deliver the message to all destinations, instead of transmitting it separately and serially.

Delirery guarantees:

If a process issues multiple independent *send* operations to individual processes, then there is no way for the implementation to provide delivery guarantees that affect the group of processes as a whole. If the sender fails half-way through sending, then some members of the group may receive the message while others do not. And the relative ordering of two messages delivered to any two group members is undefined. In the particular case of IP multicast, no ordering or reliability guarantees are in fact offered. . But stronger multicast guarantees can be made.

6.5 Consensus

Consensus is the task of getting all processes in a group to agree on some specific value based on the votes of each processes. All processes must agree upon the same value and it must be a value that was submitted by at least one of the processes (i.e., the consensus algorithm cannot just invent a value). In the most basic case, the value may be binary (0 or 1), which will allow all processes to use it to make a decision on whether to do something or not.

With election algorithms, our goal was to pick a leader. With distributed transactions, we needed to get unanimous agreement on whether to commit. These are forms of consensus. With a consensus algorithm, we need to get unanimous agreement on some value. This is a simple-sounding problem but finds a surprisingly large amount of use in distributed systems. Any algorithm that relies on multiple processes maintaining common state relies on solving the consensus problem. Some examples of places where consensus has come in useful are:

- Synchronizing replicated state machines and making sure all replicas have the same (consistent) view of system state.
- Electing a leader (e.g., for mutual exclusion)
- Distributed, fault-tolerant logging with globally consistent sequencing
- Managing group membership
- Deciding to commit or abort for distributed transactions

Consensus among processes is easy to achieve in a perfect world. For example, when we examined distributed mutual exclusion algorithms earlier, we visited a form of consensus where everybody reaches the same decision on who can access a resource. The simplest implementation was to assign a system-wide coordinator who is in charge of determining the outcome. The two-phase commit protocol is also an example of a system where we assume that the coordinator and cohorts are alive and communicating — or we can afford to wait for them to restart, indefinitely if necessary. The catch to those algorithms was that all processes had to be functioning and able to communicate with each other. Faults make it difficult. Faults include process failures and communication failures.

CHAPTER -7

REPLICATION

CHAPTER -7

REPLICATION

7.1 Replication

An important issue in distributed system is the replication of data. Data are generally replicated to enhance reliability or improve performance. Replication is the mechanism of maintaining multiple copies of data at multiple nodes. From the client's viewpoint, there is a single logical copy of data. Update at one copy of replica by a client should be reflected to all other replicas.

The main aim of replication is to provide backup on the scenario of failure. There are three reasons for replication. They are:

i. Performance enhancement
ii. Increased availability
iii. Fault tolerance

1. **Performance enhancement**

The copy of data is placed in multiple location. So, client can get the data from nearby location. This decreases the time taken to access the data. It enhances performance of the distributed system. Multiple servers located at different locations provide the same service to the client. It allows parallel processing of the client's request to the resources or computation.

Example: Web browsers store a copy of previously fetched web pages as a cached data to reduce latency of fetching resources from the server.

2. **Increased availability**

Replication is a technique for automatically maintaining the availability of data despite server failures. If data are replicated at two or more failure-independent servers, then client software may be able to access data at an alternative server should the default server fail or become unreachable. That is, the percentage of time during which the service is available can be enhanced by replicating server data. If each of n servers has independent probability p of crashing, then the availability of an object stored at each of these servers is:

1 – probability(all managers failed or unreachable) = $1 - p^n$

Example: If there is a 5% probability of any individual server failing over a given time period and if there are two servers, then the availability is $1 - 0.05^2 = 1 - 0.0025 = 99.75\%$

3. **Fault tolerance**

Replication ensures the correctness of data in addition to availability. If a server fails, the data can be accessed from other servers. If a server of a group of n servers provides faulty information, the other servers can outvote the faulty server and provide correct data to this client.

Challenges in replications:

1. **Placement of replicas**

The major challenge in replication is where to put the replicas. There are three places to put replicas.

i. **Permanent replicas**

Permanent replicas consist of cluster of servers that may be geographically dispersed.

ii. **Server initiated replicas**

Servers initiated caches include placing replicas in the hosting servers and server caches.

iii. **Client initiated replicas**

Client initiated replicas include web browsers cache.

2. **Propagation of updates among replicas**

The next challenge is to how to propagate the updates in one replica among all the replicas efficiently and faster as possible.

Push based propagation

A replica in which update occurs pushes the updates to all other replicas.

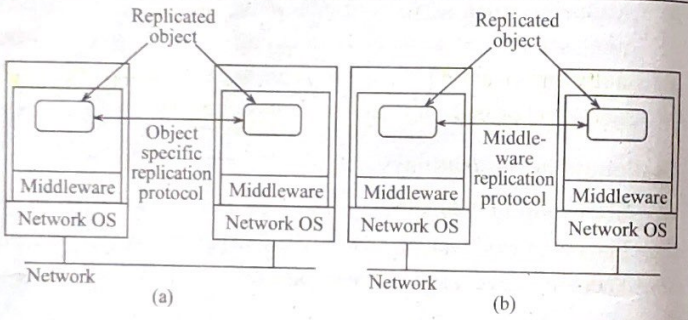
Pull based propagation

A replicas requests another replica to send the newest data it has.

3. **Lack of consistency**

If a copy is modified, the copy becomes inconsistent from the rest of the copies. It takes some time for all the copies to be consistent.

7.2 Object Replication



Approach 1: Application is responsible for replication application needs to handle consistency issues.

Approach 2: System (middleware) handles replication consistency issues are handled by the middleware.

- Simplifies application development but makes object- specific solutions harder.

7.3 Replication as Scaling Technique

Replication and caching for performance are widely applied as scaling techniques. Scalability issues generally appear in the form of performance problems. Placing copies of data close to the processes using them can improve performance through reduction of access time and thus solve scalability problems.

A possible trade-off that needs to be made is that keeping copies up to date may require more network bandwidth. A more serious problem, however, is that keeping multiple copies consistent may itself be subject to serious scalability problems. Intuitively, a collection of copies is consistent when the copies are always the same. This means that a read operation performed at any copy will always return the same result. Consequently, when an update operation is performed on one copy, the update should be propagated to all copies before a subsequent operation takes place, no matter at which copy that operation is initiated or performed. Thus we need to synchronize all replicas. In essence, this means that all replicas first need to reach an agreement on when exactly an update is to be performed locally.

On the one hand, scalability problems can be alleviated by applying replication and caching, leading to improved performance. On the other hand, to keep all copies consistent generally requires global synchronization, which is costly. In many cases, the only real solution is to loosen the consistency constraints. In other words, if we can relax the requirement that updates need to be executed as atomic operations, we may be able to avoid (instantaneous) global synchronization, and may thus gain performance.

7.4 Consistency Model

A consistency model is a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly. Normally, we expect that a read operation on a data item expects to return a value that shows the results of the last write operation on that data. In the absence of a global clock, it is difficult to define precisely which write operation is the last one.

E.g., three processes, A, B, C, and three shared variables, x,y,z originally initialized to zero in 3 replicas – each process reads values from a different replica:

x=1	y=1	z=1
print(y,z)	print(x,z)	print(x,y)

There are many possible interleavings (e.g., A1, A2, B1, B2, C1, C2; C1, B1, B2, A1, C2, A2, etc.) – the consistency model will specify what is possible…

1. Strict Consistency

This model makes use of absolute time ordering of all shared resources. It is unrealistic for distributed system as it requires absolute global time.

A data store is said to be strict consistent when it satisfies the following condition:

Any read to shared data should return value stored by the most recent write operations.

Example:

Consider a system with two processes p1 and p2. Let p1 initiate an write operation for variable x with value a. The model is strictly consistent if and only if, the read operation for variable x from both p1 and p2 just after the completion of write operation returns value a.

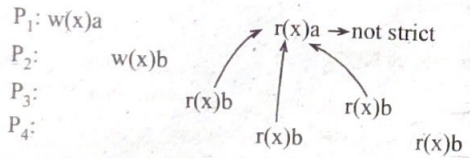


Fig.: Striet consistency

2. Sequential Consistency

Sequential consistency is an important data-centric consistency model, which was first defined by Lamport (1979) in the context of shared memory for multiprocessor systems. In general, a data store is said to be sequentially consistent when it satisfies the following condition:

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.

Example:

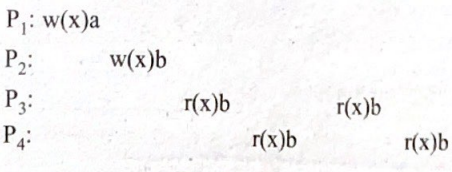


Fig.: Sequential

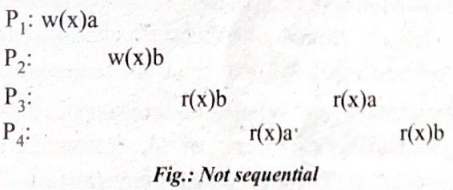


Fig.: Not sequential

3. Causal Consistency

The causal consistency model (Hutto and Ahamad, 1990) represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not. If event b is caused or influenced by an earlier event a, causality requires that everyone else first see a, then see b.

For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

Example:

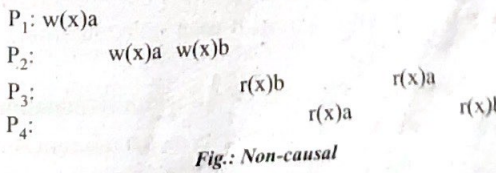


Fig.: Non-causal

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

Fig.: Causal

7.5 Fault Tolerant Services

Fault-tolerant service means a service with correct behavior despite in process/server failures, as if there was only one copy of data. A replication service is correct if it keeps responding despite faults and clients cannot tell the difference between a service provided by replication and one with a single copy of the data.

Linearizability and sequential consistency are correctness criteria for fault-tolerant services. A replicated shared object service is said to be linearizable if for any execution there is some interleaving of the series of operations issued by all the clients that satisfies the following two criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

A replicated shared object service is said to be sequentially consistent if for any execution there is some interleaving of the series of operations issued by all the clients which satisfies the following two criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

1. Passive Replication (Primary-Backup Replication)

In the passive or primary-backup model of replication for fault tolerance, there is at any one time a single primary replica

manager and one or more secondary replica managers – 'backups' or 'slaves'. In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service. The primary replica manager executes the operations and sends copies of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary.

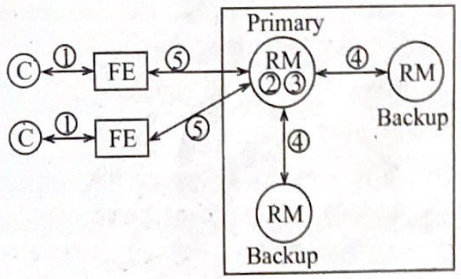


Fig.: The passive (primary-backup) model for fault tolerance

The sequence of events when a client requests an operation to be performed is as follows:

1. **Request:** The front end issues the request, containing a unique identifier, to the primary replica manager.
2. **Coordination:** The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request and if so it simply resends the response.
3. **Execution:** The primary executes the request and stores the response.
4. **Agreement:** If the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
5. **Response:** The primary responds to the front end, which hands the response back to the client.

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

Fig.: Causal

7.5 Fault Tolerant Services

Fault-tolerant service means a service with correct behavior despite in process/server failures, as if there was only one copy of data. A replication service is correct if it keeps responding despite faults and clients cannot tell the difference between a service provided by replication and one with a single copy of the data.

Linearizability and sequential consistency are correctness criteria for fault-tolerant services. A replicated shared object service is said to be linearizable if for any execution there is some interleaving of the series of operations issued by all the clients that satisfies the following two criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

A replicated shared object service is said to be sequentially consistent if for any execution there is some interleaving of the series of operations issued by all the clients which satisfies the following two criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

1. **Passive Replication (Primary-Backup Replication)**

In the passive or primary-backup model of replication for fault tolerance, there is at any one time a single primary replica

manager and one or more secondary replica managers – 'backups' or 'slaves'. In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service. The primary replica manager executes the operations and sends copies of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary.

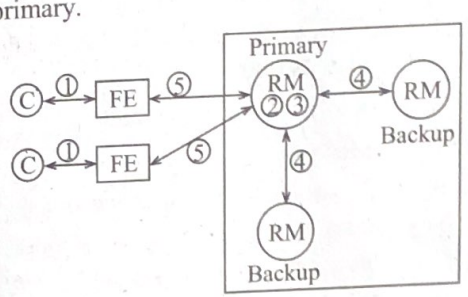


Fig.: The passive (primary-backup) model for fault tolerance

The sequence of events when a client requests an operation to be performed is as follows:

1. **Request:** The front end issues the request, containing a unique identifier, to the primary replica manager.
2. **Coordination:** The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request and if so it simply resends the response.
3. **Execution:** The primary executes the request and stores the response.
4. **Agreement:** If the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
5. **Response:** The primary responds to the front end, which hands the response back to the client.

Advantages:

i. It is simple and easy to implement.

ii. It can be implemented even if the primary replicas manger behaves non-deterministically.

Disadvantages:

i. It provides high overhead.

ii. If primary replica manager fails, more latency is incurred as new view is formed.

iii. It cannot tolerate byzantine failure.

2. **Active Replication**

In the active model of replication for fault tolerance, the replica managers are state machines that play equivalent roles and are organized as a group. Front ends multicast their requests to the group of replica managers and all the replica managers process the request independently but identically and reply. If any replica manager crashes, then this need have no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way.

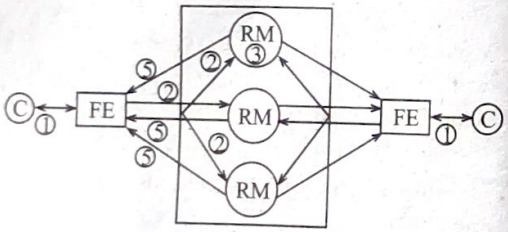


Fig.: Active replication

Under active replication, the sequence of events when a client requests an operation to be performed is as follows:

i. **Request:** The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst. It does not issue the next request until it has received a response.

ii. **Coordination:** The group communication system delivers the request to every correct replica manager in the same (total) order.

iii. **Execution:** Every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.

iv. **Agreement:** No agreement phase is needed, because of the multicast delivery semantics.

v. **Response:** Each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and on the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest (it can distinguish these from responses to other requests by examining the identifier in the response).

Advantages:

i. It can tolerate byzantine failure.

ii. If any replica manger fails, it does not affect the system performance.

Disadvantages:

i. Replica manager must be deterministic.

ii. Atomic broadcast protocol must be used.

7.6 High Availability Services

The alternatives we have seen are designed for fault tolerance. For some applications, rapid or at least reasonable response times may be required.

Fault tolerant systems send updates in an eager fashion—all correct replica manager receive updates as soon as possible. This may be unacceptable for high availability systems. It may be desirable to increase performance by providing slower (but still acceptable) updates with a minimal set of replica manager. Weaker consistency tends to require less agreement and provides more availability.

A few systems that provide highly available services:

- Gossip
- Bayou
- Coda

Gossip and Bayou both allow clients to make updates to local replicas while partitioned. In each system, replica managers exchange updates with one another when they become reconnected. Gossip provides its highest availability at the expense of relaxed, causal consistency. Bayou provides stronger eventual consistency guarantees, employing automatic conflict detection, and the technique of operational transformation to resolve conflicts. Coda is a highly available file system that uses version vectors to detect potentially conflicting updates.

The Gossip Architecture

The gossip architecture is a framework for implementing highly available services by replicating data close to the point where groups of clients need it. The name reflects the fact that the replica managers exchange 'gossip' messages periodically in order to convey the updates they have each received from clients.

A gossip service provides two basic types of operation: *queries* are read-only operations and *updates* modify but do not read the state. A key feature is that front ends send queries and updates to any replica manager they choose-any that is available and can provide reasonable response times. The system makes two guarantees, even though replica managers may be temporarily unable to communicate with one another:

1. Each client obtains a consistent service over time.
2. Relaxed consistency between replicas.

To support relaxed consistency, the gossip architecture supports causal update ordering. It also supports stronger ordering guarantees in the form of forced(total and causal) and immediate ordering. The choice of which ordering to use is left to the application designer and reflects a trade-off between consistency and operation costs.

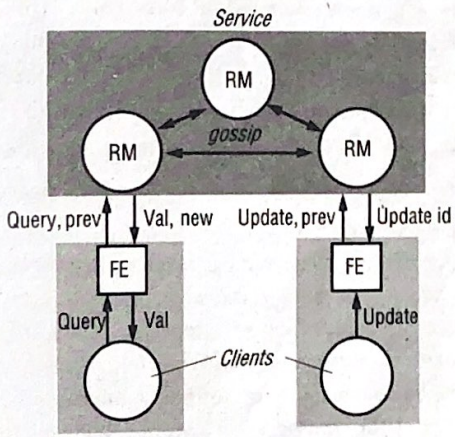


Fig.: Query and update operation in a gossip service

Gossip processing of queries and updates:

The five phases in performing a client request are:

- **Request:** The front end normally sends requests to only a single replica manager at a time. However, a front end will communicate with a different replicas manager when the one it normally uses fails or becomes unreachable, and it may try one or more others if the normal managers is heavily loaded. Front ends, and thus clients, may be blocked on query operations. The default arrangement for update operation, on the other hand, is to return to the client as soon as the operation has been passed to the front end; the front end then propagates the operation in the background. Alternatively, for increased reliability, clients

may be prevented from continuing until the update has been delivered to $f + 1$ replica managers, and so will be delivered everywhere despite up to f failures.

- **Update response:** If the request is an update then the replica manager replicas as soon as it has received the update.
- **Coordination:** The replica manager that receives a request does not process it until it can apply the request according to the required ordering constraints. This may involve receiving updates from other replica managers, in gossip messages. No other coordination between replica managers is involved.
- **Execution:** The replica manager executes the request.
- **Query response:** If the request is a query then replica manager replies at this point.
- **Agreement:** The replica manager update one another by exchanging gossip messages, which contain the most recent updates they have received. They are said to update one another in a *lazy* fashion, in that gossip messages may be exchanged only occasionally, after several updates have been collected, or when a replica manager finds out that is missing an update sent to one of its peers that it needs to process a request.

7.7 Transactions with Replicated Data

Transactions are sequences of one or more operations, applied in such a way as to enforce the ACID properties. Objects in transactional systems may be replicated to increase both availability and performance. From a client's viewpoint, a transaction on replicated objects should appear the same as one with non-replicated objects. In a non-replicated system, transactions appear to be performed one at a time in some order. This is achieved by ensuring a serially equivalent interleaving of clients' transactions. The effect of transactions performed by clients on replicated objects should be the same as if they had been

performed one at a time on a single set of objects. This property is called *one-copy serializability.*

7.7.1 Architectures for Replicated Transaction

Here, we assume that a front end sends client requests to one of the group of replica managers of a logical object. In the primary copy approach, all front ends communicate with a distinguished 'primary' replica manager to perform an operation, and that replica manger keep the backups up to date. Alternatively,front ends may communicate with any replica manager to perform an operation-but coordination between the replica managers is consequently more complex.

The replica manager that receives a request to perform an operation on a particular object is responsible for getting the cooperation of the other replica managers in the group that have copies of that object. Different replication schemes have different rules as to how many of the replica managers in a group are required for the successful completion of an operation.

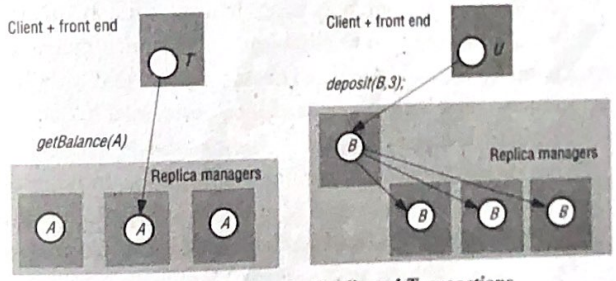


Fig.: Architectures for Replicated Transactions

i. **Read-one/write-all scheme**

In Read-one/write-all scheme, every write operation must be performed at all of the replica managers, each of which sets a write lock on the write lock on the object affected bye the operation. Each read operation is performed by a single replica manager, which set a read lock on the object affected by the operation.

ii. **Primary copy replication scheme**

In the primary copy replication scheme, all clients request are directed to a single primary replica manager. To commit a transaction, the primary communicates with the backup replica managers and then, in the eager approach, replies to the client.

iii. **Two-phase commit protocol:**

The two-phase commit protocol becomes a two-level nested two-phase commit protocol. As before, the coordinator of a transasction communication with the workers. But, if either the coordinator or a worker is a replica manager it will communicate with the other replica managers to which it passed requests during the transaction.

7.7.2 Available Copies Replication:

Simple read-one/write-all replication is not a realistic scheme, because it cannot be carried out if some of the replica managers are unavailable, either because they have crashed or because of a communication failure. The available copies scheme is designed to allow for some replica managers being temporarily unavailable. The strategy is that a client's read request on a logical object may be performed by any available replica manager but that a client's update request must be performed by all available replica managers in the group with copies of the object. The idea of the 'available members of a group of replica managers' is similar to Coda's available volume storage group.

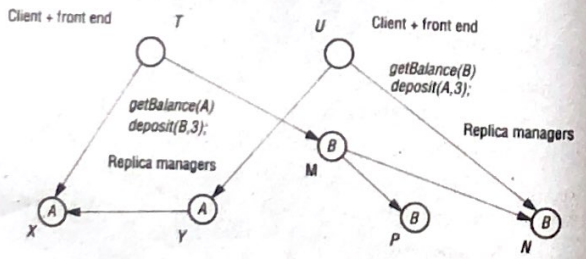


Fig.: Available Copies

In the normal case, client requests are received and performed by a functioning replica managers. Read requests can be performed by the replica manager that receives them. Write requests are performed by the receiving replica manager and all the other available replica managers in the group. For Example, in figure above, the get Balance operation of transaction T is performed by X, whereas its deposit operation is performed by M, N and P. Concurrency control at each replica mangers affects the operations performed locally.

7.7.3 Network Partition

a. **Available copies with validation**

Reads are serviced by any available replica manager.

b. **Quorum Consensus Methods**

Quorum consensus is a pessimistic approach to replicated transactions. A quorum is a subgroup of replica manager that is large enough to give it the right to carry out transactions even if some Replica Manager are not available. This limits updates to a single subset of the replica manager, which update other replica manager after a partition is corrected.

c. **Virtual partition**

This approach combines Quorum Consensus to handle partitions and Available Copies for faster read operations. A virtual partition is an abstraction of a real partition and contains a set of replica managers. Figure below (a) shows an example of a write operation that is delayed until X and V can contact either Y or Z to form a virtual partition with enough Quorum votes to write, requiring a quorum of 3 replica manager for a write. This is resolved by adding Y in figure below (b).

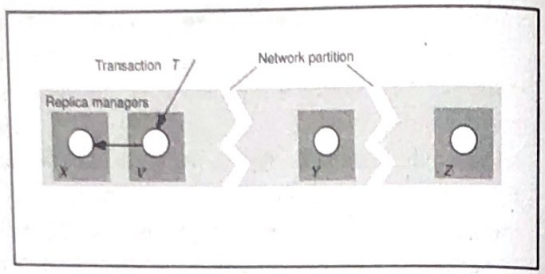


Fig. (a): Two network partitions.

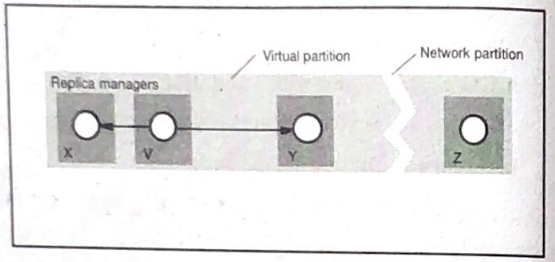


Fig. (b): Virtual partition.

One problem with virtual partitions is that if network partitions are intermittent, different virtual partitions can form, for example V1 and V2 in figure below. Overlapping virtual partitions violate one-copy serializability.

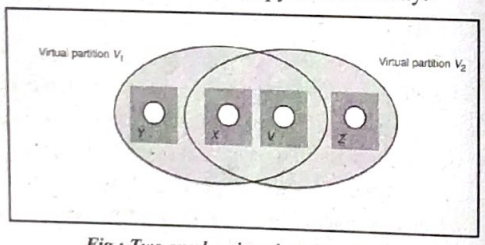


Fig.: Two overlapping virtual partitions.

CHAPTER-8

TRANSACTION AND CONCURRENCY CONTROL

CHAPTER-8

TRANSACTION AND CONCURRENCY CONTROL

8.1 Concurrency

Transaction is a sequence of requests to a server by client that ensures all the objects to remain in consistent state. The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes.

A transaction is created and managed by a coordinator which implements coordinator interface.

Operations:

openTransaction( ) → trans;
// starts new transaction and delivers unique TID trans

closeTransaction(trans) → (commit, abort);
//commit return value indicates transaction has committed
//abort return value indicates transaction has aborted

abortTransaction(trans);
//abort the transaction

Problems of concurrent transaction:

1. **Lost update problem**

Example 1:

Consider x=100

Transaction T_1	Transaction T_2	
Read(x)		X=100
X = x + 20		X=100+20=120
	Read(x)	X=100
	X=x*10	X=100*10=1000 X=120
Write(x)		X=1000
	Write(x)	Here, update by T1 is lost.

Example 2:

Assume initial balance of A, B and C be 100$, 200$ and 300$ respectively.

Transaction T:	Transaction U:
balance = b.getBalance();	balance = b.getBalance();
b.setBalance(balance*1.1);	b.setBalance(balance*1.1);
a.withdraw(balance/10);	c.withdraw(balance/10);
balance = b.getBalance(); //$200	balance = b.getBalance(); //$200
b.setBalance(balance*1.1); //$220	b.setBalance(balance*1.1); //$220
a.withdraw(balance/10); //$80	c.withdraw(balance/10); //$280

2. **Inconsistent retrieval problem**

Example: Consider initial balance of A and B be 200$ and 200$.

Transaction T:	Transaction U:
a.withdraw(100);	a. BranchbranchTotal();
b.deposit(100);	
a.withdraw(100); //$100	total = a.getBalance(); //$100
b.deposit(100); //$300	total = total + b.getBalance(); //$300

8.2 Nested Transaction

The transaction which is composed of other transaction as per requirement is called nested transaction. The outer most transaction is called top level transaction and other are called sub transaction. The sub-transactions at same level can run concurrently but their access to common objects is serialized.

CHAPTER-8

TRANSACTION AND CONCURRENCY CONTROL

8.1 Concurrency

Transaction is a sequence of requests to a server by client that ensures all the objects to remain in consistent state. The goal of transactions is to ensure that all of the objects managed by a server remain in a consistent state when they are accessed by multiple transactions and in the presence of server crashes.

A transaction is created and managed by a coordinator which implements coordinator interface.

Operations:

openTransaction( ) → trans;
// starts new transaction and delivers unique TID trans
closeTransaction(trans) → (commit, abort);
//commit return value indicates transaction has committed
//abort return value indicates transaction has aborted
abortTransaction(trans);
//abort the transaction

Problems of concurrent transaction:

1. **Lost update problem**

Example 1:

Consider x=100

Transaction T_1	Transaction T_2	
Read(x)		X=100
X = x + 20		X=100+20=120
	Read(x)	X=100
	X=x*10	X=100*10=1000 X=120
Write(x)		X=1000
	Write(x)	Here, update by T1 is lost.

Example 2:

Assume initial balance of A, B and C be 100$, 200$ and 300$ respectively.

Transaction T:	Transaction U:
balance = b.getBalance();	balance = b.getBalance();
b.setBalance(balance*1.1);	b.setBalance(balance*1.1);
a.withdraw(balance/10);	c.withdraw(balance/10);
balance = b.getBalance(); //$200	balance = b.getBalance(); //$200
b.setBalance(balance*1.1); //$220	b.setBalance(balance*1.1); //$220
a.withdraw(balance/10); //$80	c.withdraw(balance/10); //$280

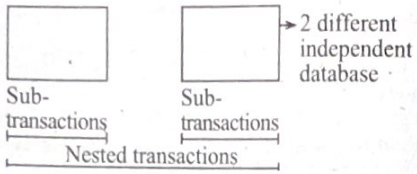
2. **Inconsistent retrieval problem**

Example: Consider initial balance of A and B be 200$ and 200$.

Transaction T:	Transaction U:
a.withdraw(100);	a. BranchbranchTotal();
b.deposit(100);	
a.withdraw(100); //$100	total = a.getBalance(); //$100
b.deposit(100); //$300	total = total + b.getBalance(); //$300

8.2 Nested Transaction

The transaction which is composed of other transaction as per requirement is called nested transaction. The outer most transaction is called top level transaction and other are called sub transaction. The sub-transactions at same level can run concurrently but their access to common objects is serialized.



Rules for commit:

1. A transaction may commit or abort only after its child transaction have completed.
2. When a sub transaction completes, it makes independent decision either to commit independent decision either to commit provisionally or to abort.
3. When a parent aborts, all of its sub transactions are aborted.
4. When a sub transactions abort, parent, can decide whether to abort or not.
5. If top level commits, all the sub transaction that have provisionally committed can commit if none of their ancestors has aborted.

8.3 Concurrency Control

It is the process of managing simultaneous execution of transactions in a shared system, to ensure the serializability of transactions.

Purpose:

- To enforce isolation
- To preserve consistency
- To resolve read write–write conflict

Concurrency control techniques

1. Locks

Transactions must be scheduled so that their effect on shared data is serially equivalent. A server can achieve serial equivalence by serializing access to the objects.

Type of locks

- Shared locks (read)
- Exclusive lock (Both read and write)

A simple example of a serializing mechanism is the use of exclusive locks. In this locking scheme, the server attempts to lock any object that is about to be used by any operation of a client's transaction. If a client requests access to an object that is already locked due to another client's transaction, the request is suspended and the client must wait until the object is unlocked. Serial equivalence requires that all of a transaction's accesses to a particular object be serialized with respect to accesses by other transactions. All pairs of conflicting operations of two transactions should be executed in the same order. To ensure this, a transaction is not allowed any new locks after it has released a lock. The first phase of each transaction is a 'growing phase', during which new locks are acquired. In the second phase, the locks are released (a 'shrinking phase'). This is called two-phase locking. For concurrent transaction reading or single transaction writing but not both; two locks are used as read and write lock.

Example:

Transaction T:	Transaction U:
balance=b.getBalance() b.setBalance(balance*1.1) a.withdraw(balance/10)	balance=b.getBalance() b.setBalance(balance*1.1) c.withdraw(balance/10)

openTansaction
balance=b.getBalance();
lock B
b.setBalance(balance*1.1);
a.withdraw(balance/10);lock A
CloseTransaction unlock A,B
	OpenTransaction
	balance=b.getBalance(); wait for T'lock on B
	b.setBalance(balance*1.1);
	lock B
	a.withdraw(balance/10); lock C
	closeTransaction

Fig. : Transaction T and U with exclusive locks

Figure illustrates the use of exclusive locks. In this example, it is assumed that when transactions T and U start, the balances of the accounts A, B and C are not yet locked. When transaction T is about to use account B, it is locked for T. When transaction U is about to use B it is still locked for T, so transaction U waits. When transaction T is committed, B is unlocked, whereupon transaction U is resumed. The use of the lock on B effectively serializes the access to B. Note that if, for example, T released the lock on B between its *getBalance* and *setBalance* operations, transaction U's *getBalance* operation on B could be interleaved between them.

2. **Two phase locking protocol**

Two phase locking protocol require both locks and unlocks being done in two phases.

Strict executions are needed to prevent dirty reads and premature writes. Under a strict execution regime, a transaction that needs to read or write an object must be delayed until other transactions that wrote the same object have committed or aborted. To enforce this rule, any locks applied during the progress of a transaction are held until the transaction commits or aborts. This is called strict two-phase locking. The presence of the locks prevents other transactions reading or writing the objects. When a transaction commits, to ensure recoverability, the locks must be held until all the objects it updated have been written to permanent storage.

Each Transaction is executed in two phases

- **Growing phase:** New locks on items can be acquired.
- **Shrinking phase:** Existing locks, but no new lock can be acquired.

The lock point is the moment when transitioning from the growing phase to the shrinking phase.

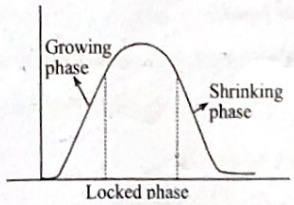


Fig: 2 phase lock

Use of locks in two-phase locking

1. When an operation accesses an object within a transaction:
   a. If the object is not already locked, it is locked and the operation proceeds.
   b. If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   c. If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   d. If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule b is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

3. **Dead lock**

The situation in which two transactions are waiting and each is dependent on the other to release a lock so as to assume is called deadlock.

Dead lock prevention

Lock all the objects used by a transaction when it starts.

Dead lock detection

- A wait-for graph is analyzed to detect deadlock by finding cycles.
- If deadlock is present, a transaction is selected for abortion.

Advantage of locks

i. Avoids the deadlocks conditions.

ii. Avoids clashes in captures the resources.

Disadvantage of locks

i. Cannot arbitrarily abort translations in times of overload or local problem such as out of memory/disk.

ii. Increased latency due to pre-processing layer that does the transaction sequencing.

iii. Reduced s concurrency.

iv. Sometimes occurs deadlocks.

8.4 Optimistic Concurrency Control

- Transaction are allowed to proceed as through there were no possibility of conflict with other clients until it completes its tusks and issues a close transaction requests.
- If conflicts arises, some transaction will be aborted and should be restarted by the client.

Each transaction has three phase.

i. Working phase

ii. Validation phase

iii. Update phase

i. Working phase

- Each transaction get copy of most recently committed version of the object.
- Read operation is performed immediately.
- Write operation records new values as tentative value.
- A same object can have several tentative values.

ii. Validation phase

- When close transaction is received, the transaction is validated to conform whether or not. There is conflicts.
- On successful validation, transaction can commit.

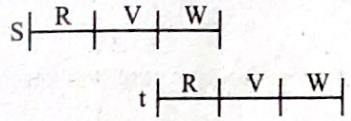
iii. Update phase

- If transaction is validated, all the tentative values are made permanent.

Validation of transaction

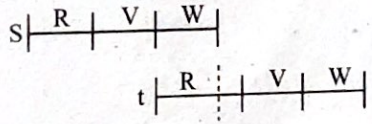
1. All transaction 'S' with earlier time stamps must have finish before transaction 't' started.

i.e.



2. If transaction 't' starts before an earlier one 'S' finishes then
   a. The set of data written by earlier transaction are not the ones reads by the current transaction.
   b. The earlier transaction completes its write phase before the current transaction enters its validation phase.

   i.e. (start (t) < finish (S) < validation)



8.5 Time Stamp Ordering

- Each transactions operation is validated when it is carried out.
- If operation cannot be validated, the transaction is aborted.
- Basic timestamp ordering rule:
  - Each transaction is assigned to write an object is valid only if that object was last read and written by earlier transactions.
  - A transactions request to read an object is valid only if that object was last written by earlier transactions.

Write Rule

If (T_c>= maximum read timestamp on D &&

T_c> write time stamp on committed versions of D)

Perform write operation on tentative version of D with write time stamp T_c.

Else

Abort T_c.

Read Rule

if (T_i write time stamp on committed version of D)

{

Let D_{sel} be version of D with max write time stamp < = T_i

if (D_{sel} is committed)

Performed read on D_{sel}.

Else

wait or abort then reapply.

}

Else

abort T_i.

8.6 Comparison of Concurrency Control Method

- Time stamp and lock use pessimistic approach.
- Time stamp is better than lock for read only transactions.
- Lock is better when operations are pre dominantly updates.
- Time stamps aborts transaction immediately.
- Locking makes the transaction wait.
- With optimistic, all transactions are allowed to proceed.

8.7 Introduction to Distributed Transaction

- Those transaction that access object managed by multiple servers is called distributed transaction.

- It requires either all of the servers involved commit the transaction or all of them abort the transaction.
- One of the servers acts as a coordinator which ensure some outcomes at all the servers.

Flat distributed transaction

- In flat distributed transaction, al client makes requests to more than one servers.
- It completes each of its requests before going on to next one, according server objects sequentially.

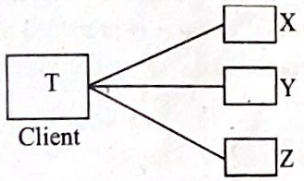


Fig: Flat transaction

Nested transaction

- In Nested transaction, the top level open sub transaction; each of which can further open sub transaction.
- The sub transaction at same level run concurrently.

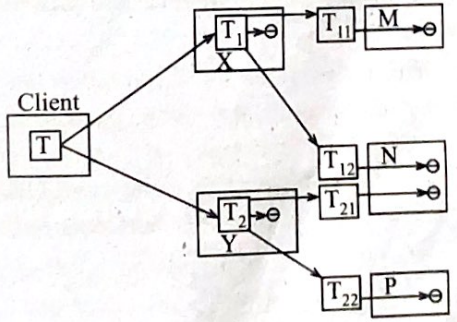


Fig: Nested distributed transaction

8.8 Atomic Commit Protocol

Two Phase Commit Protocol

The original two-phase commit protocol (2PC) is due to Gray (1978) Without loss of generality, consider a distributed

transaction involving the participation of a number of processes each running on a different machine. Assuming hat no failures occur, the protocol consists of the following two phases, each consisting of two steps:

Voting phase

1. The coordinator sends a VOTE-.REQUEST message to all participants.
2. When a participant receives a VOTE-.REQUEST message, it returns either a VOTE_COMMIT message to the coordinator telling the coordinator that it is prepared to locally commit its part of the transaction, or otherwise a VOTE-ABORT message.

Completion phase

3. The coordinator collects all votes from the participants. If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a GLOBAL_COMMIT message to all participants. However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a GLOBAL..ABORT message.
4. Each participant that voted for a commit waits for the final reaction by the coordinator. If a participant receives a GLOBAL_COMMIT message, it locally commits the transaction. Otherwise, when receiving a GLOBAL..ABORT message, the transaction is locally aborted as well.

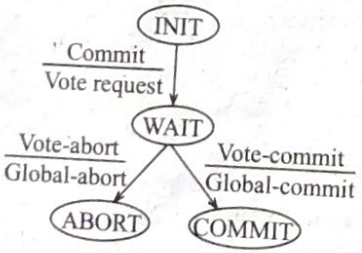


Fig: Coordinator

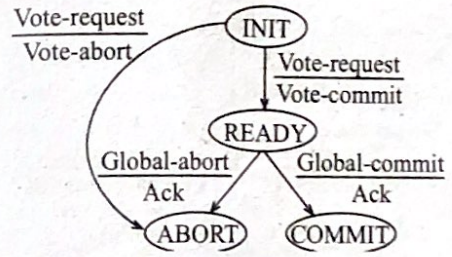


Fig: Participant

3 phase commit protocol

A problem with the two-phase commit protocol is that when the coordinator has crashed, participants may not be able to reach a final decision. Consequently, participants may need to remain blocked until the coordinator recovers. Skeen(1981) developed a variant of 2PC, called the three-phase commit protocol (3PC), that avoids blocking processes in the presence of fail-stop crashes. Although 3PC is widely referred to in the literature, it is not applied often in practice as the conditions under which 2PC blocks rarely occur. We discuss the protocol, as it provides further insight into solving fault-tolerance problems in distributed systems.

Like 2PC, 3PC is also formulated in terms of a coordinator and a number of participants. Their respective finite state machines are shown in The essence of the protocol is that the states of the coordinator and each participant satisfy the following two conditions:

1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.

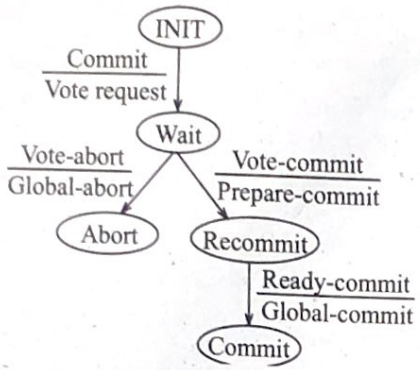


Fig: Co-ordinator

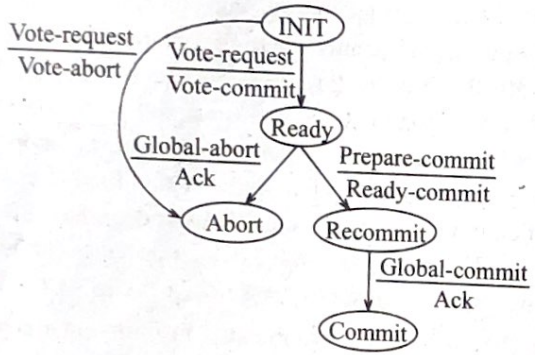


Fig: Participator

Co-ordinator, sends vote- request message to all the participants, after which it waits for incoming response.

If any participant votes to abort the transactions the final decision will be to abort as well, so the coordinator sends global - abort.

However the transaction can be committed, a prepare-commit message is sent.

Only after each participants has acknowledge, it is now prepared to commit, will the coordinator send the final commit message by which the transaction is actually committed.

8.9 The Lost Update Problems

The lost update problem occurs when two transactions that access the same system items have their operations interleaved in a way that makes the value of some system itself incorrect:

i.e. Interleaved use of the same data items would cause same problem when an update operations from one transaction overwrites update from a second transaction.

Example: N=5, m=4

Time	T_1	T_2	Value
1	Read_item(x)		x = 80
2	x: = x–n		x = 80 – 5 = 75 (which is not written in db)
3		Read_ item (x)	x = 80; (T_2 still reads in the original value of x, the updated value of x is lost
4		x: = x+m	x = 80 + 4 = 84
5	Write_item(x)		x = 75 is written in dB
6	Read_item(y)		
7		Write_ item (x)	x=84 over writes x=75 a wrong record is written in dB.
8	y= y + N		
9	write_item (x)		

The final result should be x = 80 // 75 + 4 = 79 but in the concurrent operation of the figure below it is 84 because the update that canceled 5 seats in T_1 was lost.

8.10 Distributed Deadlock Avoidance

As in centralized system, distributed deadlock avoidance handles deadlock prior to occurrence. Additionally, in distributed system, transaction location and transaction, control issues needs to be addressed. Due to the distributed nature of the transaction, the following conflicts may over.

1. Conflicts between tow transaction in the same site.
2. Conflicts between two transaction in the different sites.

In case of conflict,

Ones of the transactions may be aborted or allowed to wait as per distributed wait die algorithm.

Let us assume that, there are 2 transaction T_1& T_2 arrives at site p and tries to lock a data item which is already locked by t_2 at that site. Hence, there is a conflict at site P.

The algorithm are as follows:

1. **Distributed wound – die**

   If T_1 is older than T_2, T_1 is allowed to wait. T_1 can resume execution after site P receives a message that T_2 has either s committed or aborted successfully at all sites. If T_1 is younger than T_2, T_1 is aborted.

2. **Distributed wait - wait**

   - If T_1 is than T_2, T_2 needs to be aborted.
   - If T_1 is younger than T_2, T_1 is allowed to wait.

CHAPTER-9

FAULT TOLERANCE

CHAPTER-9

FAULT TOLERANCE

9.1 Introduction to Fault Tolerance

Fault means defect within hardware or software. *Fault tolerance* is defined as the characteristic by which a system can mask the occurrence and recovery from failures. In other words, a system is said to be *fault tolerant* if it can continue to operate in the presence of failures.

A system is said to be k-fault tolerant system if it is able to function properly even if k-nodes of the system suffers from concurrent failures.

Requirements of Fault Tolerant System

i. **Availability**

Availability is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users. In other words, a highly available system is one that will most likely be working at a given instant in time.

ii. **Reliability**

Reliability refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly-reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability. If a system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable. Similarly, a system that never crashes but is shut down for two weeks every August has high reliability but only 96 percent availability. The two are not the same.

3. Safety

Safety refers to the situation that when a system temporarily fails to operate correctly, nothing catastrophic happens. For example, many process control systems, such as those used for controlling nuclear power plants or sending people into space, are required to provide a high degree of safety. If such control systems temporarily fail for only a very brief moment, the effects could be disastrous.

4. **Maintainability**

Maintainability refers to how easy a failed system can be repaired. A highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically.

5. **Security**

Avoidance or tolerance of deliberate attacks to the system.

Types of fault:

i. **Node fault**

A fault that occurred at an individual node participating in the distributed system.

Example: A machine in a distributed system fails due to some error in database configuration of that machine.

ii. **Program fault**

A fault that occurred due to some logical or syntactical errors in the code.

Example: A program that is designed to filter the data by category but instead filtered by other means.

iii. **Communication fault**

A fault that occurred due to unreliable communication channels connecting the node.

Example: A node sending data "010110" to another node but due to some problem in communication channel, the receiver receives "010111".

iv. Timing fault

The fault that occurred due to mismatch on timing of any particular response.

Example: A server replies too late or a server is provided with data too soon that is has no enough buffer to hold the data.

9.2 Process Resilience

Process resilience is a mechanism to protect against faulty processes by replicating and distributing computations in a group. The group can be of two types:

i. Flat group

- All the processes within a group have equal roles.
- Control is completely distributed to all processes.
- Good for fault tolerance as information is exchanged immediately
- Impose more overhead
- Difficult to implement

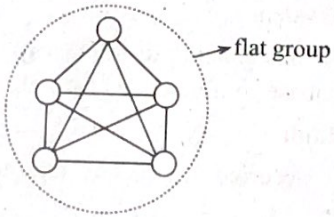


Fig.: Communication in a flat group

ii. Hierarchical group

- All the communications are handled by a single process designated as a coordinator.
- Not completely fault tolerant and scalable
- Easy to implement

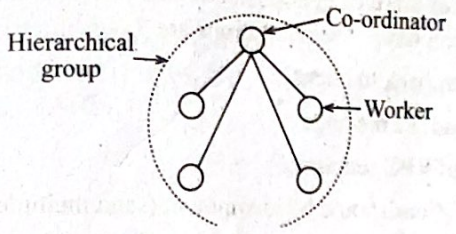


Fig.: Communication in a simple hierarchical group

Advantages and disadvantages of flat group and hierarchical group:

Flat group: It is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. For example, to decide anything, a vote often has to be taken, incurring some delay and overhead.

Hierarchical group: It has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else.

9.3 Reliable Client-Server Communication

In many cases, fault tolerance in distributed systems concentrates on faulty processes. However, we also need to consider communication failures because a communication channel may exhibit crash, omission, timing, and arbitrary failures.

Five different classes of failures that can occur are:

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

Each of these categories poses different problems and requires different solutions. These solutions are:

1. Report back to client
2. Resend the message
3. Use of RPC semantics
4. Operations should be idempotent (send multiple requests)
5. Kill the orphan computation

9.4 Distributed Commit

Distributed commit deals with methods to ensure that either all the processes commit to the final result or none of them do. This ensures consistency of the distributed system.

Distributed commit is often established by means of a coordinator. In a simple scheme, this coordinator tells all other processes that are also involved, called participants, whether or not to (locally) perform the operation in question. This scheme is referred to as a *one-phase commit protocol*. It has the obvious drawback that if one of the participants cannot actually perform the operation, there is no way to tell the coordinator.

In practice, a more sophisticated schemes are needed, the most common one being the *two-phase commit protocol*. But again the main drawback of this protocol is that it cannot efficiently handle the failure of the coordinator. That's why, a *three-phase commit protocol* has been developed.

9.4.1 Two-Phase Commit

- The client who initiated the computation acts as coordinator.
- Processes required to commit act as participant.

Phase

1. a. Coordinator sends VOTE_REQUEST to participants.
   b. When participants receive VOTE_REQUEST, it reply YES or NO. If it sendsNO, it aborts its local computations.

2. a. Coordinator collects all votes. If all YES, it sends commit to all participants. Else, it sends abort.
   b. Each participant waits for COMMIT or ABORT and handles accordingly.

Algorithm

Coordinator:

1. Multicast VOTE_REQ to all participants
2. If all votes are not collected, repeat:
   - Wait for any incoming vote
   - If timeout, write GLOBAL_COMMIT and multicast to all participants.
3. If all participants send COMMIT, write GLOBAL_COMMIT and multicast to all participants; else write, GLOBAL_ABORT and multicast to all participants.

Participants

1. Wait for VOTE_REQ from coordinator
2. If timeout, write VOTE_ABORT and exit.
3. If participant votes ABORT, write VOTE_ABORT and send ABORT to coordinator.
4. If participant votes COMMIT
   - Write VOTE_COMMIT and sendCOMMIT to coordinator
   - Wait for DECISION from coordinator
   - If time out, multicast DECISION_REQ to other participants, Waits for decision and write decision.
   - If coordinator DECISION is COMMIT write GLOBAL_COMMIT else write GLOBAL_ABORT .

Handling decision requests

1. **Repeat until True:**
   - Wait until any incoming DECISION_REQ is received.

- Reads most recently recorded state from local log.
- If state is GLOBAL_COMMIT send GLOBAL_COMMIT.
- Else if state is GLOBAL_ABORT or INIT, send GLOBAL _ABORT.
- Else, skip.

9.4.1 Snapshot algorithm

The responsibilities of the initiator process and of all other processes are stated below.

1. **Initiator process P_i.**
   i. take a check point
   ii. Increment the checkpoint sequence no.
   iii. Send a checkpoint request $<C_P,i>$ to successor and predecessor.
   iv. Continue with normal condition.
2. **At process P_j**
   - It receives the checkpoint $<C_p,i>$
     if checkpoint request $<C_P,i>$ is the second request received.
   - Discard the request.
   - Continue normal execution
     ELSE,
   - If serving a higher priority procedure take a checkpoint after the procedure ends,
   - Increment the checkpoint sequence no. forward the checkpoint request $<C_P,i>$ to adjacent process from which checkpoint request was not received.
   - Continue normal execution.

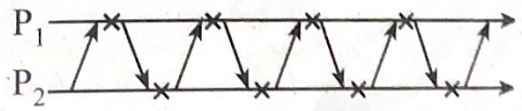
   ELSE,
   - Take a checkpoint
   - Increment the checkpoint sequence no. forward the checkpoint request $<C_P,i>$ to adjacent process from which checkpoint request was not received.

9.5 Distributed Recovery

Distributed recovery is the mechanism to handles failure. It helps to recover correct state of the system after failure.

Techniques

1. **Independent checkpoint**
   - Each process periodically checkpoints independent of other processes.
   - Upon failures, locate a consistent cut backgrounds.
   - Needs to rollbacks until consistent cut is found.



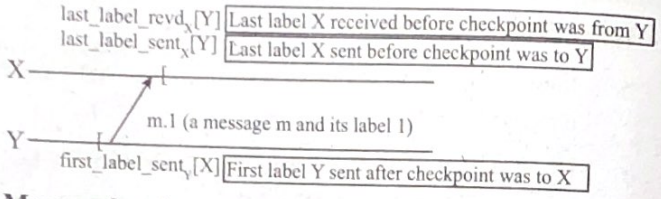
2. **Coordinated checkpoint**

A checkpointing protocol in distributed systems can be coordinated, independent or quasi-synchronous. Coordinated Checkpointing is an attractive checkpointing strategy as it is domino-free and requires a minimum number of checkpoints (maximum two checkpoints) to be stored per process on stable storage. In the coordinated checkpointing, all processes take checkpoints cooperatively and in a synchronized way. It is, therefore, also known as "Synchronous checkpointing". In this scheme the synchronization of processes is required for checkpointing and so, extra check pointing messages are required to be exchanged between processes.

Also, the underlying computation may have to be frozen. Once a synchronous checkpoint of the distributed computation has been recorded on stable storage, it can be used to recover from any future fault. If a fault occurs, all processes roll back and restart from the global consistent state given by this checkpoint. A global state of a distributed system contains one checkpoint of each process and is consistent if it does not contain any orphan messages.

An orphan message is a message, whose receiving has been recorded by the destination process in its checkpoint but whose sending is not recorded by the sender in its checkpoint.

Example:



3. Message logging

Message logging is a common technique used to build systems that can tolerate process crash failures. These protocols require that each process periodically records its local state and log the messages received since recording that state. When a process crashes, a new process is created in its place: the new process is given the appropriate recorded local state, and then it replays the logged messages in the order they were originally received.

All message-logging protocols require that the state of a recovered process be consistent with the states of the other processes. This consistency requirement is usually expressed in terms of orphan processes, which are surviving processes, whose state is inconsistent with the recovered state of a crashed process. Therefore, the big question is how our implementation of message logging will guarantee that after recovery no process is orphan.

Types of failures

1. Crash failure : Halts, but is working correctly until it halts.
2. misaim failure: fails to responds, to incoming requests.
3. Timing failure: response lies outside a specified time internal.
4. Response failure: Response is incorrect.
5. Arbitrary failure: may produce arbitrary response at arbitrary times.

Failures making by redundancy

If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failure from other processes. The key techniques for masking faults is to use redundancy.

Three kinds are possible.

1. **Information redundancy:**

   Extra bits are added to allow recovery from garbled bits. Example: Hamming code.

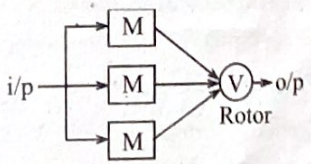
2. **Time redundancy**

   An action is performed and ten it need be, it performed again.

   Example: If a transaction aborts, it can reduce with no harm.

3. **Physical redundancy**

   Extra equipment or processes are added to make it possible for the system as a whole to tolerance the loss or malfunctioning of same components.



- It could be expended to NMR.
- The rotor produces correct output if there are no failure in the rotor and if there are no failures in 2 of the 3 models.

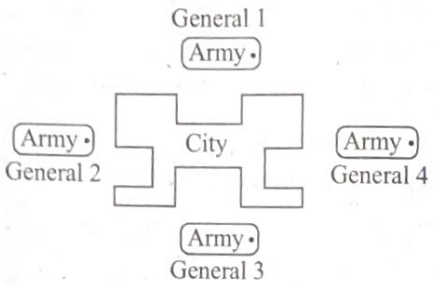
9.6 Byzantine Generals Problems

The faculty node can also generate arbitrary data pretending to be a correct one and making fault tolerant difficult.

Assumptions

1. All loyal generals must agree on the same plan of action.
2. Generals can communicate only by message passing.
3. Traitors may be anything they wish.

Problem definition



- A commanding generals must send an order to his n-1 impudent generals such that:
  - All loyal lieutenants obey the same order.
  - If commanding generals is loyal, then every loyal lieutenants obeys the order he sends.

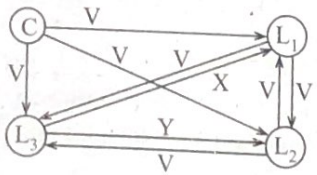
Conclusions

With m faculty processes, agreements is possible only if 2m+1 processes functions correctly.

Proof

Consider 4 generals out of which 1 is traitor as per the conclusion the agreement should be made even in case of the presence of the traitor.

Let C, L_1, L_2 and L_3 be the generals.



At the end of stage 1:

L_1–v

L_2–v

L_3–v

At the end of stage 2

L_1–v, v, x

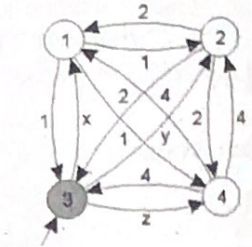
L_2–v, v, y

L_3–v, v, v

This shows that all lieutenants agree on the same actions as all of them come to consensus with majority votes of message 'v' at the end of the process.

Example:

Byzantine Generals problem



1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(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)

(c)

we illustrate the working of the algorithm for the case of N = 4 and k = 1. For these parameters, the algorithm operates in four steps.

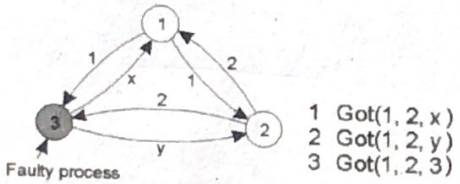
Step 1: Every nonfaulty process i sends Vi to every other process using reliable unicasting. Faulty processes may send anything. Moreover, because we are using multicasting, they may send different values to different processes. Let Vi =i. In Fig. above (a) we see that process 1 reports 1, process 2 reports 2, process 3 lies to everyone, giving x, y, and z, respectively, and process 4 reports a value of 4.

Step 2 : The results of the announcements of step 1 are collected together in the form of the vectors of Fig. above (b).

Step 3: consists of every process passing its vector from Fig. above (b) to every other process. In this way, every process gets three vectors, one from every other process. Here, too, process 3 lies, inventing 12 new values, a through 1. The results of step 3 are shown in Fig. above (c).

Step 4: each process examines the ith element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. If no value has a majority, the corresponding element of the result vector is marked UNKNOWN. From Fig. (c) we see that 1, 2, and 4 all come to agreement on the values for VI, v 2, and v 4, which is the correct result. What these processes conclude regarding v 3 cannot be decided, but is also irrelevant. The goal of Byzantine agreement is that consensus is reached on the value for the nonfaulty processes only.

Byzantine Generals Problem Example



for N = 3 and k = 1, that is, only two nonfaulty process and one faulty one, as illustrated in Fig. above. Here we see that in Fig. above (c) neither of the correctly behaving processes sees a majority for element 1, element 2, or element 3, so all of them are marked UNKNOWN. The algorithm has failed to produce agreement.

In their paper, Lamport et a1. (1982) proved that in a system with k faulty processes, agreement can be achieved only if 2k + 1 correctly functioning processes are present, for a total of 3k + 1. Put in slightly different terms, agreement is possible only if more than two-thirds of the processes are working properly.

CHAPTER-10

CASE STUDIES

CHAPTER-10

CASE STUDIES

10.1 CORBA

Goals:

The OMG's goal was to adopt distributed object systems that utilize object-oriented programming for distributed systems. Systems to be built on heterogeneous hardware, networks, operating systems and programming languages. The distributed objects would be implemented in various programming languages and still be able to communicate with each other.

Features:

CORBA consists of a language independent RMI Consists of a set of generic services useful for distributed applications. The CORBA RMI acts as a "universal translator" that permits client processes to invoke a method or process that may reside on a different operating system or hardware, or implemented via a different programming language.

The CORBA RMI consists of the following main components:

- An interface definition language (IDL)
- An architecture (discussed in Structure)
- The General Inter-ORB Protocol (GIOP)
- The Internet Inter-ORB Protocol (IIOP)

CORBA IDL Features:

Provides an interface consisting of a name and a set of methods that a client can request.

IDL supports fifteen primitive types, constructed types and a special type called Object.

- Primitive types: short, long, unsigned short, unsigned long, float, double, char, boolean, octet, and any.

- Constructed types such as arrays and sequences must be defined using type defs and passed by value.

Interfaces and other IDL type definitions can be grouped into logical units called modules.

GIOP and IIOP Features:

- **GIOP:** General Inter-ORB Protocol are the standards (included in CORBA 2.0), which enable implementations to communicate with each other regardless of who developed it.
- **IIOP:** Internet Inter-ORB Protocol is an implementation of GIOP that uses the TCP/IP protocol for the Internet.

CORBA Services:

Set of generic service specifications useful for distributed applications.

- CORBA Naming Service - essential to any ORB
- CORBA Event Service - define interfaces
- CORBA Notification Service - extension of event service
- CORBA Security Service - controls access
- CORBA Trading Service - allows location by attribute
- CORBA Transaction and Concurrency Control Service
- CORBA Persistent Object Service

CORBA RMI Structure:

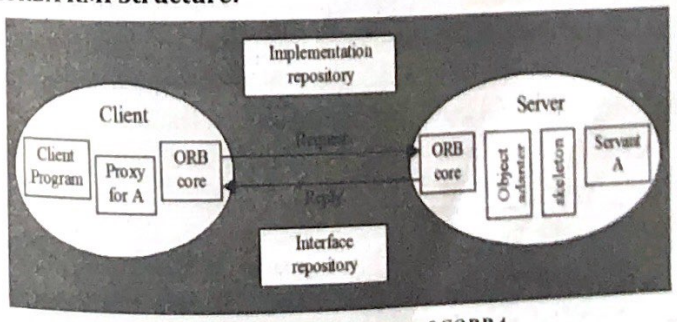


Fig.: Architecture of COBRA

ORB core

- Carries out the request-reply protocol between client and server.
- Provide operations that enable process to be started and stopped.
- Provide operations to convert between remote object references and strings.

Object Adapter (server)

- Bridges the gap between CORBA objects and the programming language interfaces of the servant classes.
- Creates remoter object references for the CORBA objects
- Dispatches each RMI to the appropriate servant class via a skeleton, and activates objects.
- Assigns a unique name to itself and each object
- Called the Portable Object Adapter in CORBA 2.0
  - Processes can run on ORB's produced by different developers.

Skeletons (server)

- An IDL compiler generates skeleton classes in the server's language.
- Dispatch RMI's to the appropriate servant class.

Client Proxies/Stubs

- Generated by an IDL compiler in the client language.
- A proxy class is created for object oriented languages
- Stub procedures are created for procedural languages.

(Both are responsible for marshalling and unmarshalling arguments, results and exceptions.)

Structure repository:

- **Implementation Repository**

  Activates registered servers on demand and locates servers that are currently running.

- **Interface Repository**

  Provides information about registered IDL interfaces to the clients and servers that require it. Optional for static invocation; required for dynamic invocation.

Application of CORBA:

- Used primarily as a remote method invocation of a distributed client – server system.
- Can communicate between clients and servers on different operating systems and implemented by different programming languages (Java cannot do this).
- Has many standards and services useful in implementing distributed applications.
- Process can be both server and client to another server
- Ideal for a heterogeneous distributed system like the Internet

10.2 Mach

- Mach is a microkernel that provides the most elementary services needed for an operating system.
- First version: 1986 for VAX 11/784, a fovr, cpu multiprocessor.
- Shortly thereafter, part to IBM PC/RT and sun B were done.
- By 1987 mach was also running on the encore and sequent multiprocessor.
- As 1988, the mach 25 kernel was large and monolithic, due to presence of large amount of Berkeley's code.
- CMU removed all Berkeley code from the kernel and put it user spaced.

Goals

- Support diverse architectures
- Function with varying inter computer networks speeds
- Simplified kernels structure
- Distributed operation

- Integrated memory management & IPC
- Heterogeneous system support

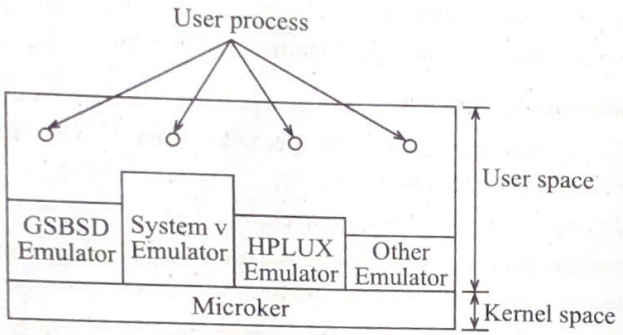


Fig: Mach micro kernel

- Trans part multiprocessing: avoiding issues in BSD
- Protected message passing: Better than Unix message messaging
- Extensible microkernel
- Multiple levels of operating system
- Other's O/s's implemented as application.
- Basis for Next O/s, Mac x o/s, OSF/s.

10.3 TIB/ Rendezvous

TIB makes it easy to create distributed applications that exchange data across a network. It supports many hardware and software platform like TIB/ Rendezvous API, TIB/ Rendezvous daemon.

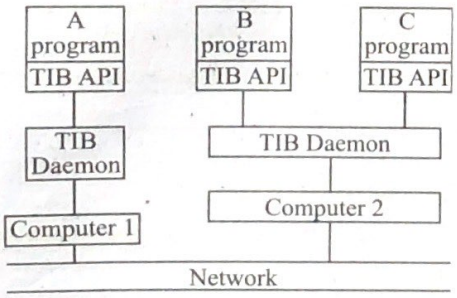


Fig: Component of TIB

TIB Daemon

- Transmits outbound message from program processes to the network
- Delivers inbound message from the network to program processes
- Filters subject- address messages
- Delivers messages reliably

TIB/ Rendezvous language interface

- The C/C++ API

  It support widest array of hardware platform
- Com and .Net

  It run on windows platform
- Java

  Programs can be stand- alone java application or browser - based java applets
- A pearls loadable module presents a TIB/Rendezvous API that is parallel to the ANSIc.

10.4 JINI

- Jini technology allows devices to dynamically establish communication to share and exchange service across a network.
- Provides simple mechanisms which enable device to plug together to form an impromptu community.
- Jini is an extension of java, i.e. java consists of one virtual machine, so, jini is a kind of virtual network.

Services

- Jini system consists of services that can be collected to performs a particular task.
- Services can be dynamically added from the system as needed.

- Services communicate using a service protocol which is a set of interface written in Java.

Look up services

- Major point of contact between users of the system and the system.
- Find and adds services to jini federation.
- Maps functionally to a set of objects in jini.
- Services added using two protocols.
  - Discovery: locates appropriate lookup service.
  - Join-join lookup service.

Jini system

	Infrastructure	Programming model	Services
Java	Java virtual machine Java remote method invocation	Beans swing graphics toolkit	Enterprises java beams, java naming & directory services. Java transaction services
Jini	Discovery look up services	Leasing transactional distributed events	Java spaces transaction manager.

❀ ❀ ❀