

DISTRIBUTED SYSTEM (ELECTIVE)

VI C.S.

DISTRIBUTED SYSTEMS

1

PREVIOUS YEARS QUESTIONS

PART-A

Prob.1 What is Middleware layer? [RTU 2011]

Sol. Services in Distributed Systems

- **Heterogeneity and Distributed computing** problems are simplified by many vendors as they are offering distributed system services that have standard programming interfaces and protocols.
- **Standard programming interfaces** make it easier to execute applications on variety of platforms, providing the user freedom from dependency on vendor specific products.
- **Standard protocols** providing the services in Distributed System enable programs to interoperate.

These distributed system services are called **middleware** because they sit "in the middle," layering above the operating system and networking software and below industry-specific applications.

Prob.2 Mention few examples of distributed systems.

Sol. Web search, Massively multiplayer online games (MMOGs), Financial Trading markets, SOA based systems etc.

Prob.3 List the key design goals of distributed system.

Sol. The Key Design Goals of Distributed System :

1. Reliability
2. Scalability
3. Consistency
4. Security
5. Performance

Prob.4 Write the difference between Network OS and Distributed OS.

Sol. Difference between Network OS and Distributed OS.

Item	Distributed OS	Networks OS
Multiproc.	Very high	High
Degree of transparency	Yes	Yes
Same OS on all nodes	Yes	No
Number of copies of OS	1	N
Basis for communication	Shared memory	Messages
Resources management	Global central	Global distributed
Scalability	No	Moderately
Openness	Closed	Closed
		Open

Prob.5 What is clock skew and clock drift?

Sol. Clock skew (offset) : The difference between the time on two clocks.

Clock drift : They count time at different rates. Ordinary quartz clocks drift by ~ 1 sec in 11-12 days (10^{-6} secs/sec). High precision quartz clocks drift rate is ~ 10^{-7} or 10^{-8} secs/sec.

PART-B

Prob.6 What do you mean by distributed computing environment (DCE). Explain its features, services and goals. [RTU 2016, 2017]

DS-2

OR

Write short note on Distributed Computing Environment. [R.T.U. 2018, 2017]

Sol. The DCE supplies a framework and toolkit for developing client/server applications.

The framework includes a remote procedure call (RPC) mechanism known as DCE/RPC, a naming (directory) service, a time service, an authentication service and a distributed file system (DFS) known as DCE/DFS. It provides security services to protect and control access to data, name services that make it easy to find distributed resources, and a highly scalable model for organizing widely scattered users, services, and data. DCE runs on all major computing platforms and is designed to support distributed applications in heterogeneous hardware and software environments. DCE is a key technology in three of today's most important areas of computing: security, the World Wide Web, and distributed objects.

Distributed Computing Environment Characteristics and Features

1. **Strongly-typed Interfaces:** A major problem in a large scale distributed environment is ensuring the consistency and integrity of the messages and methods shared senders and receivers. If the type signatures of these messages and methods become inconsistent, the reliability and correctness of a distributed system will be severely compromised.
2. **Security:** Distributed applications are generally more vulnerable to security breaches than are stand-alone applications, since there are more access points for an intruder to attack. For example, most shared-media networks (such as Ethernet, token ring, and FDDI) provide only limited built-in protection against cable tapping.
3. **Object Location and Selection:** DCC frameworks generally provide location brokers that allow clients to access remote object services via higher level names (rather than by low-level memory addresses or IP/port numbers), and traders that allow remote objects to be selected based on the desired characteristics of the services they provide. Location brokers and traders simplify distributed system administration and promote more flexible and dynamic placement of services throughout a network by automatic distributed object selection.
4. **Multiple Entities :** In the distributed computing environment many entities (user or subsystem which composes the distributed system) can participate in the system.
5. **Concurrency :** Different components of distributed system may run concurrently as the components may be loosely coupled.
6. **Resource Sharing :** Entities in distributed system can share resources with each other.

B.Tech. (VI Sem.) C.S. Solved Papers

7. **Openness :** Underlying architecture, protocols and infrastructure can be extended or replaced without affecting the system behavior.

Services Provided by Distributed System: Fundamental distributed services provide tools for software developers to create the end-user services needed for distributed computing such as Remote Procedure Call, Directory Service, Time service, and Thread services etc.

Data-sharing services provide end users with capabilities built upon the fundamental distributed services. These services require no programming on the part of the end user and facilitate better use of information such as distributed file system and diskless support.

Goals

1. **Security:** The DCE security server authenticates all users and servers, ensuring that people, servers, and applications are who they claim to be. DCE authorization capabilities can protect distributed resources from unauthorized use.

2. **Lower Maintenance Costs:** The use of formal interface definitions by DCE RPC allows any number of programmers to write applications that communicate correctly, while keeping versions and configurations in sync. The interface definition can be maintained and versioned, and the version number compiled into the application will ensure that each client finds a compatible server.

3. **Scalability and Availability:** The use of replication for both security and directory servers aids in network response time, as well as service availability. DCE installations in the tens and hundreds of thousands of users operate successfully today.

Prob.7 What are the design issues in Cooperative Autonomous systems? Discuss. [RTU 2014]

Sol. A cooperative autonomous system is a high-level service-oriented software system that requires the support of communication mechanisms on which higher-level communication protocols are built. As an example, imagine how a real estate transaction can be accomplished in a cooperative autonomous system. The house buyer, which is a client process, can make a request to buy a house either directly from the house owner or indirectly through a real estate agent (both are considered server processes). The owner is a client process to the realtor. Realtors can form a real estate agency, a larger server that can refer a house buyer to an appropriate realtor server. The seller is a client to the real estate agency as well as a client to the realtor. The buyer can locate the real estate agency by looking it up in the yellow pages, which

- Fault tolerance and security
- Security threats and failures are both system faults.
- The problem of failures can be alleviated if there is redundancy in the system - problem with checkpointing.
- Security: authentication and authorization.

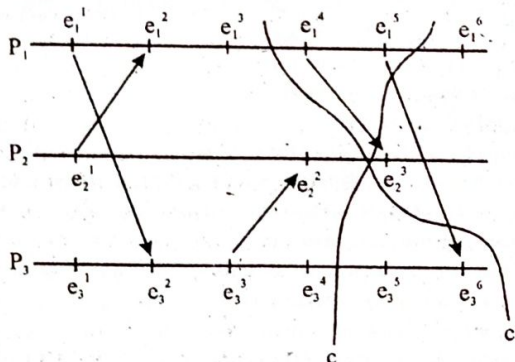
Prob.8 How will you ensure the consistency of a cut in state recording of the various process? Explain.

[RTU 2014]

Sol. Consistency of a Cut: Casual precedence happen to be the appropriate formalism for distinguishing the two classes of cut exemplified by c and c' . A cut c is consistent if for all events e and e' .

$$(e \in c) \wedge (e' \rightarrow e) \rightarrow e' \in c$$

In other words, a consistent cut is left closed under the casual precedence relation. In graphical representation, if all arrows that intersect the cut have their bases to the left and heads to the right of it, then the cut is consistent, otherwise it is inconsistent.



Here cut c is consistent and c' is inconsistent.

Prob.9 What is the significance of marker in Chandy - Lamport algorithm? Explain.

[RTU 2014]

OR

What do you mean by state recording of distributed system? Explain your answer using Chandy-Lamports algorithm.

[RTU 2012]

OR

Explain Chandy - Lamport algorithm for consistent state recording.

[RTU 2013]

Sol. Chandy-Lamport Algorithm

- The Chandy-Lamport algorithm uses a control message, called a marker whose role in a FIFO system is to separate messages in the channels.

- After a site has recorded its snapshot, it sends a marker, along all of its outgoing channels before sending out any more messages.

- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.

- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

- The algorithm can be initiated by any process by executing the "Marker Sending Rule" by which it records its local state and sends a marker on each outgoing channel.

- A process executes the "Marker Receiving Rule" on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the "Marker Sending Rule" to record its local state.

- The algorithm terminates after each process has received a marker on all of its incoming channels.

- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Chandy-Lamport algorithm

Marker Sending Rule for process i

Process i records its state.

For each outgoing channel C on which a marker has not been sent, i sends a marker along C before i sends further messages along C . Marker Receiving Rule for process j on receiving a marker along channel C : if j has not recorded its state then record the state of C as the empty set follow the "Marker Sending Rule" else

Record the state of C as the set of messages received along C after j 's state was recorded and before j received the marker along C

Correctness and Complexity

Correctness : Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition C2 is satisfied.

When a process p_i receives message m_{ij} that precedes the marker on channel C_{ij} , it acts as follows: If process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij} . Thus, condition C1 is satisfied.

Complexity : The recording part of a single instance of the algorithm requires $O(e)$ messages and $O(d)$ time, where e is the number of edges in the network and d is the diameter of the network.

Prob.10 Explain Distributed Computing and its Paradigms.

[RTU 2013]

Sol. Distributed Computing : Distributed computing is a field of computer science that studies distributed systems. A distributed system is a software system in which components located on networked computers to communicate and

Prob.11 Explain various types of operating systems associated with the distributed systems along with design issues in distributed operating systems.

OR
[R.TU.2019]

Write and explain various theoretical issues in distributed systems.
OR
[R.TU.2019]

Explain different types of operating systems in detail.
OR
[R.TU.2019]

What is theoretical issues in distributed system?
OR
[R.TU.2017]

Explain design issues in distributed operating system.
[R.TU.2019]

Sol. An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs. Application programs usually require an operating system to function. For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware. Operating systems can be found on almost any device that contains a computer from cellular phones and video game consoles to supercomputers and web servers.

Types of Operating Systems
Real-time Operating Systems : In this, Response Time is already fixed. Real-time operating systems are used to control machinery, scientific instruments and industrial systems means time to Display the Results after Processing has fixed by the Processor or CPU. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time, every time it occurs.

Single-user, Single Task : As the name implies, this operating system is designed to manage the computer so that one user can effectively do one thing at a time. The Palm OS for Palm handheld computers is a good example of a modern single-user, single-task operating system.

Single-user, Multi Tasking : This is the type of operating system most people use on their desktop and laptop computers today. Microsoft's Windows and Apple's MacOS platforms are both examples of operating systems that will let a single user have several programs in operation at the

same time. For example, it's entirely possible for a Windows user to be writing a note in a word processor while downloading a file from the Internet while printing the text of an email message.

Multi-user : A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. The operating system must make sure that each of the programs they are using has sufficient and separate resources so that a problem with one user does not affect the entire community of users. Unix, VMS and many other operating systems, such as AFS, are examples of multi-user operating systems.

Distributed Operating System : In a distributed system, software and data may be distributed around the system. programs and files may be stored on different storage devices which are located in different geographical locations and may be accessed from different computer terminals.

Batch Processing Operating System : In a batch processing operating system interaction between the user and processor is limited or there is no interaction at all during the execution of work. Data and programs that need to be processed are bundled and collected as a batch and executed together.

Batch processing operating systems are ideal in situations where:

- There are large amounts of data to be processed
- Similar data needs to be processed
- Similar processing is involved when executing the data

The system is capable of identifying times when the processor is idle at which time 'batches' may be processed. Processing is all performed automatically without any user intervention.

Parallel Operating Systems : Parallel operating systems are able to use software to manage all of the different resources of the computers running in parallel, such as memory, caches, storage space, and processing power. Parallel operating systems also allow a user to directly interface with all of the computers in the network.

Multiprocessing : Generally a computer has a single processor. Multiprocessing means a computer has a just one CPU for processing the instructions, but if we are running multiple jobs then this will decrease the speed of CPU. For increasing the speed of processing then we use the multiprocessing. In the multiprocessing there are two or more CPU in a single operating system if one CPU will fail, then other CPU is used for providing backup to the first CPU. With the help of multiprocessing, we can execute many jobs at a time. All the operations are divided into the number of CPUs. If first CPU completed his work before the second CPU, then the work of second CPU will be divided into the first and second

Design Issues in Distributed Operating Systems : There are some issues that arise in distributed systems:

1. **Openness** : One of the important features of distributed systems is openness and flexibility, every service is equally accessible to every client (local or remote). It is easy to implement, install and debug new services, users can write and install their own services. This feature makes it easier to build and change. In Symmetric Kernel systems, calls are served by the kernel, e.g., UNIX. Microkernel provides minimal services like I/O, some memory management, some low-level process management and scheduling, low-level I/O (e.g., Network support, multiple file systems, multiple system interfaces). It needs standard interface and communication protocols for implementation.
2. **Security** : Encryption can be used to provide adequate protection of shared resources and to keep sensitive information secret when transmitted in messages over a network. Security for information resources has three components:
 - (i) Confidentiality (protection against disclosure to unauthorized individuals)
 - (ii) Integrity (protection against corruption)
 - (iii) Availability (protection against interference with the means to access the resources)

Distributed systems should allow communication between programs/users/resources on different computers. Security risks are associated with free access to all of resources. The appropriate use of resources by different users has to be guaranteed.

3. **Scalability** : A system is described as scalable if the system will remain effective when there is a significant increase in the number of resources and in the number of users. The architecture and the implementation must allow it. The algorithms must be efficient under the circumstances to be expected. For example, the Internet. Scalability help in controlling the cost of physical resources and in controlling performance loss. It also helps in preventing software resources running out and in avoiding performance bottlenecks.

4. **Failure handling** : Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. Failures in a distributed system are partial—that is, some components fail while others continue to function. Therefore the handling of failures is particularly difficult.

5. **Concurrency** : Both services and applications provide resources that can be shared by clients in a distributed system. The process that manages a shared resource could take one

client request at a time. But that approach limit throughput. Therefore services and applications generally allow multiple client requests to be processed concurrently. The presence of multiple users in a distributed system is a source of concurrent requests to its resources. So, any object that represents a shared resource in a distributed system must be responsible for ensuring that it operates correctly in a concurrent environment.

6. **Transparency** : Distributed systems should be perceived by users and application programmers as a whole rather than as a collection of cooperating components.

Transparency has different dimensions that were identified by ANSI. It represents various properties that distributed systems should have. How to achieve the single-system image, i.e., how to make a collection of computers appear as a single computer. Hiding all the distribution from the users as well as the application programs can be achieved at two levels:

- At a lower level, make the system look transparent to programs. Both requires uniform interfaces such as access to files, communication.

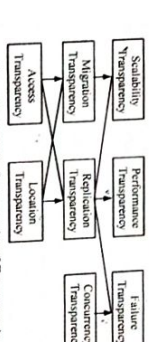


Fig. : Diagrammatic Representation of Transparencies

Table : Different forms of transparency in a distributed system

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location while in use
Replication	Hide that a resource may be shared by several concurrent users
Concurrency	Hide that a resource may be shared by several competitive users
Failures	Hide the failures and recovery of a resource
Persistence	Hide whether a resource is in memory or is on disk

(a) **Access Transparency** : Enables local and remote resources to be accessed using identical operations.

(b) **Location Transparency** : Enables resources to be accessed without knowledge of their location.

(c) **Concurrency Transparency** : Enables several processes to operate concurrently using shared resources without interference between them.

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

(f) **Mobility Transparency** : Allows the movement of resources and clients within a system without affecting the operation of users or programs.

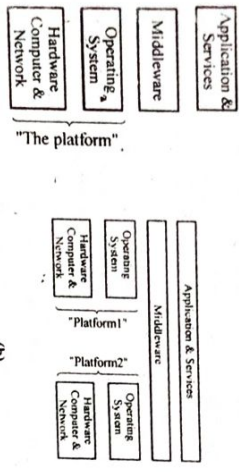
(g) **Performance Transparency** : Allows the system to be reconfigured to improve performance as loads vary.

(h) **Scaling Transparency** : Allows the system and applications to expand in scale without change to the system structure or the application algorithms.

7. **Quality of service** : Once users are provided with the functionality that they require of a service, such as the file service in a distributed system, we can go on to ask about the quality of the service provided. The main non-functional properties of systems that affect the quality of the service experienced by clients and users are reliability, security and performance. Adaptability to meet changing system configurations and resource availability has been recognized as a further important aspect of service quality.

8. **Reliability** : Distributed system should be more reliable than a single system. Availability is fraction of time the system is usable. Redundancy improves it. Like it needs to maintain consistency, need to be secure and should be fault tolerance, need to mask failures, recover from errors. One of the main goals of building distributed systems is improvement of reliability. If machines go down, the system should work with the reduced amount of resources. There should be a very small number of critical resources, resources which have to be up in order the distributed system to work. Key pieces of hardware and software (critical resources) should be replicated if one of them fails another one takes up. Data on the system must not be lost, and copies stored redundantly on different servers must be kept consistent. The more copies kept, the better the availability, but keeping consistency becomes more difficult.

9. Performance : Without gain on this, why bother with distributed systems.



- Fig. : (a) Software Architecture, (b) Same Looking at two Distributed Nodes
- Performance loss due to communication delays :
 - fine-grain parallelism: high degree of interaction
 - coarse-grain parallelism
 - Performance loss due to making the system fault tolerant

Pr.Q.12 Explain Goals and Services of distributed systems
[R.T.U. 2019]

OR
What are the goals behind developing distributed systems?
[RTU 2015]

Explain.
Sol. A distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, 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. Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications.

Four goals that should be met to make a distributed system worth the effort are:

1. It should make resources (printers, computers, storage facilities, data, files, Web pages, and networks, etc.) easily accessible.
2. It should reasonably hide the fact that resources are distributed across a network, so that the users perceive it as a single system.
3. It should be open.
4. It should be scalable.

1. Making Resources Accessible

Main goal of distributed system is to make it easy for the users (and applications) to access remote resources and to share them in a controlled and efficient way. Sharing resources helps in performance enhancements and is also economic. Accessibility essentially means the opportunity to

use available hardware, software or data anywhere in the system. A resource manager is a software module that manages a resource of a particular type. Various resource managers control access, offer a scheme for naming, and control concurrency. A resource sharing model describes how resources are made available how can they be used and how the service provider and user interact with each other.

Models for Resource Sharing:

- Client-server resource model
 - Server processes act as resource managers, and offer services (collection of procedures).
 - Client processes send requests to servers.
- Object-based resource model
 - Any entity in a process is modelled as an object with a message based interface that provides access to its operations.
 - Any shared resource is modelled as an object.

Issues in Resource Sharing:

Contention: A conflict over access to a shared resource such as random access memory, disk storage, cache memory, internal buses or external network devices. Resolving resource contention problems is one of the basic functions of operating systems. Various low-level mechanisms can be used to aid this, including locks, semaphores, mutexes and queues. The other techniques that can be applied by the operating systems include intelligent scheduling, application mapping decision, and page colouring.

2. Distribution Transparency

To hide the fact that its processes and resources are physically distributed across multiple computers – systems should be transparent. Different forms of transparency in a distributed system (ISO, 1995) is given below:

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed.
Location	Hide where a resource is located.
Migration	Hide that a resource may move to another location.
Relocation	Hide that a resource may be moved to another location while in use.
Replication	Hide that a resource is replicated.
Concurrency	Hide that a resource may be shared by several competitive users.
Failure	Hide the failure and recovery of a resource.

Issues in transparency:

- **Timing:** E.g. requesting an electronic newspaper to appear in your mailbox before 7 A.M. local time, as usual, while you are currently at the other end of the world living in a different time zone.

Synchronization: E.g. a wide-area distributed system that connects a process in San Francisco to a process in Amsterdam limited by laws of physics – a message sent from one process to the other takes about 35 milliseconds. It takes several hundreds of milliseconds not using a computer network as signal transmission is not only limited by the speed of light, but also by limited processing capacities of the intermediate switches.

Performance: E.g. many Internet applications repeatedly try to contact a server before finally giving up. Consequently, attempting to mask a transient server failure before trying another one may slow down the system as a whole.

Consistency: E.g. need to guarantee that several replicas, located on different continents, need to be consistent all the time – a single update operation may now even take seconds to complete, something that cannot be hidden from users.

Context Awareness: E.g. notion of location and context awareness is becoming increasingly important, it may be best to actually expose distribution rather than trying to hide it. Consider an office worker who wants to print a file from her notebook computer, it is better to send the print job to a busy nearby printer, rather than to an idle one at corporate headquarters in a different country.

Limits of Possibility: Recognizing that full distribution transparency is simply impossible, we should ask ourselves whether it is even wise to pretend that we can achieve it.

3. Openness

The aim here is to offer services according to standard rules that describe the syntax and semantics of those services. The system should be able to interact with services from other open systems, irrespective of the underlying environment.

- Systems should conform to well-defined interfaces.
- Systems should support portability of applications.
- Systems should easily inter-operate a property of a product or system, whose interfaces are completely understood, to work with other products or systems, present or future, without any restricted access or implementation).

Openness can primarily be achieved by making the system independent from the heterogeneity of Hardware, Platforms and Languages and by using well-defined interfaces.

In Computer networks, there is a standard set of rules that governs the format, contents, and meaning of messages sent and received, whereas in a Distributed systems, the services are specified through interfaces (which

also help achieve openness), which are often described in an Interface Definition Language (IDL).

Interface Definitions written in an IDL nearly always capture only the syntax of services. Specify names of the available functions with types of parameters, return values, possible exceptions that can be raised, etc.

- Allows an arbitrary process that needs a certain interface to talk to another process that provides that interface.
- Allows two independent parties to build completely different implementations of those interfaces, leading to two separate distributed systems that operate in exactly the same way.

Goals: An open distributed system should also be extensible. i.e.

- It should be easy to configure the system out of different components (possibly from different developers).
- It should be easy to add new components or replace existing ones without affecting those components that stay in place.

4. Scalability

Scalability of a system is measured with respect to:
• **Size** - This can easily add more users and resources to the system.

• **Geographic extent** - A geographically scalable system is one in which the users and resources may lie far apart.

• **Administrative scalability** - Can be easy to manage even if it spans many independent administrative organizations.

Scalability Limitations of Size	Concept	Example
Centralized services	A single server for all users.	
Centralized data	A single on-line telephone book.	
Centralized algorithms	Doing routing based on complete information.	

Geographical scalability Limitations

- E.g. currently hard to scale existing distributed systems designed for local-area networks is that they are based on synchronous communication.
- A client requesting service blocks until a reply is sent back.
- Works fine in LANs where communication between two machines is generally at worst a few hundred microseconds.

DS.10

- In a wide-area system, inter-process communication may be hundreds of milliseconds, three orders of magnitude slower.

Unreliability of communication:

- Communication in wide-area networks is inherently unreliable and point-to-point.
- Local-area networks provide reliable communication based on broadcasting, making it much easier to develop distributed systems. For example, consider the problem of locating a service.
- E.g. in a local-area system, a process can broadcast a message to every machine, asking if it is running the service it needs.
- Only those machines that have that service respond, each providing its network address in the reply message.
- Such a location scheme is unthinkable in a wide-area system: just imagine what would happen if we tried to locate a service this way in the Internet.

Services of Distributed Systems : Modern distributed systems feature a set of basic services, which can be complemented with specific services depending on the objectives of the system. The most relevant generic services of a distributed system are summarized in table. In recent systems, the security aspects have been considerably reinforced, leading to the consolidation of security-related functions- obviously including any basic registration, authentication and authorization services- in a global Security Service. High quality-of-service (QoS) communication has also become increasingly important due the need to achieve greater dependability, timeliness and security of communications on open systems. This is stressed everyday by the emergence of demanding applications, in the electronic business, cooperative (CSCW, teleconference), and multimedia rendering do-mains (games and movies). It would not be surprising to see certain advanced communication services appear bundled in distributed system support packages, such as: reliable group communication; real time communication; cryptographic communication.

Table : Generic Distributed System Services

Name Service	Based on a replicated and distributed database, supplies the global names and addresses of users, services and resources.
Registration Authentication and Authorization Services	Registers users and services, performs runtime authentication of users and control of their access to services and resources.

B.Tech. (VI Sem.) C.S. Solved Papers

File Service	Provides the abstraction of a unique file system, globally accessible, made of distributed repositories, eventually replicated for performance or availability.
Networking Service	Provides access by users and programs to the basic networking and communication facilities (e.g. sockets over TCP/IP on LAN, dial-up, Internet).
Remote Invocation Service	Provides for remote operation client-server invocation.
Brokerage Service	Performs trading and binding of services and users in a heterogeneous environment (e.g., Object Request Broker)
Time Service	Supplies and keeps synchronized a global time reference, normally made of local clocks.
Administration Service	Performs tactical management tasks, in order to manage users and keep the system resources and services operating correctly.

Prob.13 (a) State and explain the challenges of distributed system.

(b) Explain Architecture models.

[R.T.U. 2017]

OR

Explain Architectural models for distributed system along with the challenges of distributed systems. [R.T.U. 2018]

Sol.(a) Challenges of Distributed System : The scalable distributed systems has the following challenges :

(1) Scalability :

(i) Controlling the Cost of Physical Resources :

As the demand for a resource grows, it should be possible to extend the system, at reasonable cost, to meet it. It must be possible to add server computers to avoid the performance bottlenecks that would arise if a single file server had to handle all file access requests.

(ii) Controlling the Performance Loss : We consider the set of data whose size is proportional to the number of resources or users in the system. In that case algorithm that use hierarchical structures scale more better than those that use linear structures.

(iii) Preventing Software Resources Running Out :
An example of lack scalability is shown by the number

A specialization of clusters where a load balancer is used to direct incoming traffic to one of many worker nodes. It predates grid computing and doesn't rely on a homogenous abstraction of the underlying nodes as much as Grid computing. A web farm tends to have very specialized machines dedicated to each component type and is far more optimized for that specific task.		In a grid, each node is relatively independent of others; problems are solved in a divide and conquer fashion.
Systems are tightly coupled	Systems are loosely coupled	Offers a set of services to users, it acts as a sort of supercomputer by sharing processing power across the machines.
Ex: Sony PlayStation clusters and Microsoft Xbox clusters	Ex: Amazon Web Services (AWS), Google App Engine	Systems are loosely coupled (a loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components. Sub-areas include the coupling of classes, interfaces, data, and services.) Ex: Future Grid

Prob.19 Explain DCE architecture model and its components in detail. (RTU 2013)

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

DCE is based on three distributed computing models:

Client/Server

A way of organizing a distributed application

Shared Files

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

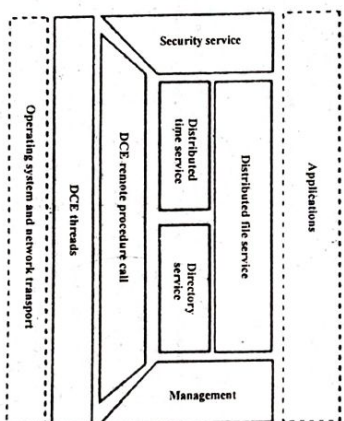


Fig. : DCE architecture

Remote Procedure call
A way of communicating between parts of a distributed application.

One way of implementing communications between a client and a server of a distributed application is to use the procedure call model. In this model, the client makes what

Distributed System

looks like a procedure call, and waits for a reply from the server. The procedure call is translated into network communications by the underlying RPC mechanism. The server receives a request and executes the procedure, returning the results to the client.

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

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

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

Directory Service

The DCE Directory Service is a central repository for information about resources in the distributed system. Typical resources are users, machines and RPC-based services. The information consists of the name of the resource and its associated attributes. Typical attributes could include a user's home directory, or the location of an RPC-based server.

The DCE Directory Service consists of several parts: the Cell Directory Service (CDS), the Global Directory Service (GDS), the Global Directory Agent (GDA) and a Directory Service programming interface. The CDS manages a database of information about the resources in a group of machines called a DCE cell. The Global Directory Service implements an international, standard directory service and provides a global namespace that connects the local DCE cells into one worldwide hierarchy. The GDA acts as a go-between for cell and global directory services. Both CDS and GDS are accessed using a single Directory Service application programming interface (API).

Security Service

There are three aspects to DCE security: authentication, secure communications, and authorization. They are implemented by several services and facilities that together

comprise the DCE Security Service. These include the Registry Service, the Authentication Service, the Privilege Service, the Access Control List (ACL) Facility, and the Login Facility.

The identity of a DCE user or service is authenticated by the Authentication Service. Communications are protected by the integration of DCE RPC with the Security Service, by the integration over the network can be checked for tampering or encrypted for privacy. Finally, access to resources is controlled by comparing the credentials conferred to a user by the Privilege Service with the rights to the resource, which are specified in the resource's Access Control List. The Login Facility initializes a user's security environment, and the Registry Service manages the information (such as user passwords) in the DCE Security database.

Time Service

The DCE Time Service (DTS) provides synchronized time on the computers participating in a Distributed Computing Environment. DTS synchronizes a DCE host's time with Coordinated Universal Time (UTC) an international time standard. DTS cannot keep the time in each machine precisely the same but can maintain it to a known accuracy. DTS also provides services which return a time range to an application (rather than a single time value), and which compare time ranges from different machines. They can be used to schedule and synchronize events across the network.

File Service

The DCE File Service (DFS) allows users to access and share files stored on a File Server anywhere on the network, without having to know the physical location of the file. Files are part of a single, global namespace. A user anywhere on a network can access any file, just by knowing its name. The File Service achieves high performance particularly through caching of file system data. Many users can access files that are located on a given File Server without a large amount of network traffic or delays.

Note: The File Service is based on a personal computer view of files, and is not relevant to the CICS Transaction Server for z/OS environment.

Threads

DCE Threads supports the creation, management, and synchronization of multiple threads of control within a single system layer, the layer below DCE. If the host operating system already supports threads, DCE can use that software and DCE Threads is not necessary. Because all operating systems do not provide a threads facility and DCE components require threads to be present, this user-level threads package is included in DCE.

CONCURRENT PROCESSES AND PROGRAMMING 2

PREVIOUS YEARS QUESTIONS

PART-A

Prob.1 What is threads?

Sol. Threads are an efficient way to improve application performance through parallelism. Each thread of a process has its own program counter, its own register states and its own stack. But all the threads shares the same address space. Threads are often referred as lightweight process.

Prob.2 What are the main advantages of using threads instead of multiple processes?

Sol. Advantages of Threads over Multiple Processes :

1. **Context Switching :** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For eg. : they require space to store, the PC, the SP, and the general-purpose registers but they do not require space to share memory information. Information about open files of I/O devices in use, etc.
2. **Sharing :** Threads allow the sharing of lot of resources that cannot be shared in process, for example, sharing code section, data section, OS resources like open file.

Prob.3 What are the design issues for RPC?

Sol. The Design Issues of RPC :

- The style of programming promoted by RPC programming with interfaces.
- The key issue of transparency and how it relates to remote procedure calls.

Prob.4 What is Name Service?

Sol. A name service stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote such as users, computers, services and objects.

Prob.5 What is Directory Service?

Sol. A service that stores collection of bindings between names and attributes and that looks up entries that match attribute-based specification is called a directory service.

PART-B

Prob.6 Explain Graph models for process representation in detail.

(R.T.U. 2019)

Sol. Graph Models for Process Representation : Having described processes and threads, we are now interested in how they are put together. Processes are related by their need for synchronization and/or communication. Synchronization means that the execution of some processes must be serialized in a certain order. A special example of synchronization is the precedence relationship between two processes when execution of one process must precede the other. In many cases, the precedence relationship or the ordering of processes is not always necessary for communicating processes as long as long as messages can be properly exchanged. Figure 1 uses graph models to illustrate these two different views of process interactions. The synchronous process graph in the Direct Acyclic Graph (DAG) model shows explicit precedence relationships and a partial ordering of a set of processes. The undirected edges in the asynchronous process graph indicate the existence of communication paths, and thus some dependency between processes.

Distributed System

The graphs in figure 1 are too abstract to model any nontrivial system behavior. Yet they are not completely useless. For instance, we can use them to model the mapping between processes and processors in a distributed system. The synchronous process graph can be used to analyze the makespan (total completion time) of a set of cooperating processes, and the asynchronous communication graph can be used to study processor allocation for optimizing interprocessor communication overhead. A more detailed graph of distributed processes often makes analytical analysis more difficult and sometime even intractable. There is a trade-off between the expressive and analytical powers of the system model.

In figure 1, the directed edges in the precedence graph can be interpreted as a synchronous communication of a sent or received message. Results produced by a process are passed to a successor process as input. The communication transaction happens and is synchronized only at the completion of a process and the beginning of its successor process(es). This communication is more precisely specified than in the asynchronous process graph, where nothing is said about how and when communication occurs except that is a communication path between the processes.

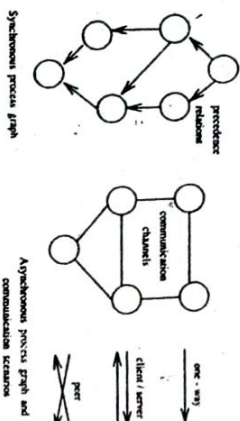


Fig. 1: Graph models for process interaction

Processes in the undirected graph live indefinitely, as opposed to the directed graph where processes have a finite lifetime. There are three types of communication scenarios for the asynchronous communication graph model: one-way, client/server and peer. Using communication terminology, they correspond to simplex, half-duplex, and full-duplex communication, respectively. An application process in one-way communication sends messages and expects no reply. A good example is the broadcast of information. Client/server communication is two-way; it makes a request and receives a reply. Many communication applications use this assumption, which is a master/slave relationship. Peer communication is a symmetrical two-way exchange of messages. It is commonly used for coordination among peer processes.

Both synchronous and asynchronous process graph models are suitable for performance analysis but lack sufficient detail to describe the interaction among processes. Figure 2

shows the time-space model, a better representation of communication and precedence relations. The existence of the events and actual communications are explicit in the model. The corresponding precedence or communication graph can be derived easily from the time-space diagram. This model possesses more information for analyzing the instantaneous interactions among processes than for evaluating the overall system performance. The choice of models depends on the intended analysis.

Whether processes are represented by a precedence or communication graph, the interaction of processes must be expressed in a language or by some other formal mechanism. We can invent a new concurrent language for concurrent processing, but it is easier to take an existing sequential language and extend it with additional language constructs or operating system supports to provide for the creation, communication, and synchronization of processes.

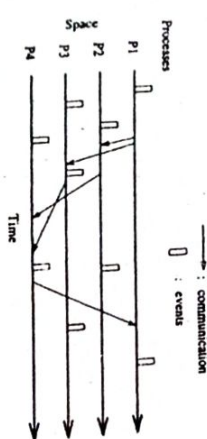


Fig. 2: A time-space model for interacting processes

For example, we can introduce a cobegin/coend control structure or use fork/join system calls to create and synchronize concurrent processes. Processes created in this way are more tightly coupled since they have a master/slave or parent/child relationship and share some common attributes. They also work cooperatively toward a common goal and are normally implemented by a single individual or organization. This approach is suited to the implementation of the precedence process graph model. The other scenario for structuring processes is to assume a peer-to-peer relationship among processes. Processes interact with one another only through interprocess communication. No precedence relation exists between processes. In fact they may be created independently, executed asynchronously, and have different life spans. This is best modeled by a communication process graph. In this case, the specification and interaction of processes become an operating system issue rather than a language issue.

Prob.7 Explain the concept of RPC and RMI's in distributed systems.

OR

(R.T.U. 2019)

Is there any difference between RPC and RMI? Explain.

[RTU 2014]

Sol. RPC (Remote Procedure Call) and RMI (Remote Method Invocation) are two mechanisms that allow the user to invoke or call processes that will run on a different computer from the one the user is using. The main difference between the two is the approach or paradigm used. RMI uses an object oriented paradigm where the user needs to know the object and the method of the object he needs to invoke. In comparison, RPC isn't object oriented and doesn't deal with objects. Rather, it calls specific subroutines that are already established.

RPC is a relatively old protocol that is based on the C language, thus inheriting its paradigm. With RPC, you get a procedure call that looks pretty much like a local call. RPC handles the complexities involved with passing the call from the local to the remote computer. RMI does the very same thing; handling the complexities of passing along the invocation from the local to the remote computer. But instead of passing a procedural call, RMI passes a reference to the object and the method that is being called. RMI was developed by Java and uses its virtual machine. Its use is therefore exclusive to Java applications for calling methods on remote computers.

In the end, RPC and RMI are just two means of achieving the same exact thing. It all comes down to what language you are using and which paradigm you are used to. Using the object oriented RMI is the better approach between the two, especially with larger programs as it provides a cleaner code that is easier to track down once something goes wrong. Use of RPC is still widely accepted, especially when any of the alternative remote procedural protocols are not an option.

1. RMI is object oriented while RPC isn't
2. RPC is C based while RMI is Java only
3. RMI invokes methods while RPC invokes functions
4. RPC is antiquated while RMI is the future

Prob.8 Explain the concept of processes and threads.

[R.T.U. 2018]

Sol. Processes : Process is an executing instance of a program. For example, When you double click on the Google Chrome icon on your computer, you start a process which will run the Google Chrome program. When you double click on a notepad icon on your computer, a process is started that will run the notepad program.

A process is sometimes referred as **active entity** as it resides on the primary memory and leaves the memory if the system is rebooted. Several processes may be related to same program. For example, you can run multiple instances of a notepad program. Each instance is referred as a process.

Threads : Thread is the smallest executable unit of a process. For example, when you run a notepad program, operating system creates a process and starts the execution of main thread of that process.

A process can have multiple threads. Each thread will have their own task and own path of execution in a process. For example, in a notepad program, one thread will be taking user inputs and another thread will be printing a document.

All threads of the same process share memory of that process. As threads of the same process share the same memory, communication between the threads is fast.

Processes and threads can be represented like below:

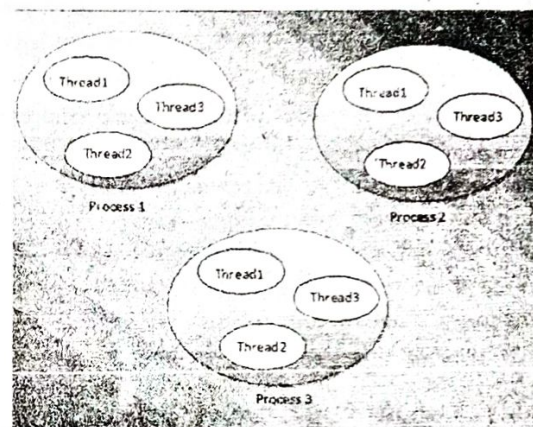


Fig.

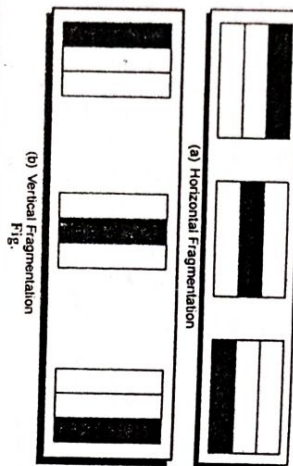
Process Vs Thread : Below is the list of differences between process vs thread.

S. No.	Process	Thread
1.	Processes are heavy weight operations.	Threads are light weight operations.
2.	Every process has its own memory space.	Threads use the memory of the process they belong to.
3.	Inter process communication is slow as processes have different memory address.	Inter thread communication is fast as threads of the same process share the same memory address of the process they belong to.
4.	Context switching between the process is more expensive.	Context switching between threads of the same process is less expensive.
5.	Processes don't share the memory with other processes.	Threads share the memory with other threads of the same process.

Prob.9 State the characteristics of a concurrent programming language.

[RTU 2015]

DS.25



Prob.13 What do you understand by 'Transaction Communication'? Explain. [RTU 2014]

Sol. Transaction Communication

The service-oriented request/reply communication and multicast combined to form a new level of communication called *transaction communication*. A transaction is more commonly known as a fundamental unit of interaction between client and server processes in a database system. A database transaction is represented by a *sequence of synchronous request/reply operations* that satisfy the atomicity, consistency, isolation, and durability (ACID) properties. Transactions in communication are similar to transactions in databases except they are defined as a set of *asynchronous request/reply communications* that also have the ACID properties but are without the sequential constraint of the operations in a database transaction. A communication transaction may involve the multicast of the same message to replicated servers and different requests to partitioned servers. Transaction services and replicated servers and different requests to partitioned servers.

The ACID Properties

The ACID properties are primarily concerned with achieving the *concurrency transparency* goal of a distributed system. Concurrency transparency is a property that allows sharing of objects without interference. In a sense, the execution of a transaction appears to take place in a critical section. However, operations from different transactions are interleaved (in some "safe" way) to gain more concurrency. In addition, transactions have additional properties:

- Atomicity:** Either all of the operations in a transaction are performed or none of them are, in spite of failures.
- Consistency:** The execution of interleaved transactions is equivalent to a serial execution of the transactions in some order.
- Isolation:** Partial results of an incomplete transaction are not visible to others before the transaction is successfully committed.

DS.26

Durability: The system guarantees that the results of a committed transaction will be made permanent even if a failure occurs after the commitment.

Since all four properties are related to *consistency*, sometimes it is preferable to call the second property *serializability* to differentiate it from the others. Atomicity refers to the consistency of replicated or partitioned objects. Violation of isolation is seeing something that has never occurred, and violation of durability is not seeing something that has actually occurred. Both are inconsistent perceptions of the system state.

The Two-phase Commit Protocol

The two-phase commit (2PC) protocol is analogous to a real-life unanimous voting scheme. Voting is initiated by the coordinator of a transaction. All participants in the distributed transaction must come to an agreement about whether to commit or abort the transaction and must wait for the announcement of the decision. Before a participant can vote to commit a transaction, it must be prepared to perform the commit. A transaction is committed only if all participants agree and are ready to commit.

Each participant (including the coordinator) maintains a private work space for keeping track of updated data objects. Each update contains the old and new value of a data object. Updates will not be made permanent until the transaction is finally committed, to ensure the isolation semantics of the transactions. To cope with failures, it is necessary to flush the updates to a stable storage. The updates recorded in the stable storage are an *activity log*. The transaction. Each participating site has an *activity log*. The activity log can be replayed upon the recovery of a failure to facilitate either the *redo* of committed transactions or the *undo* of uncommitted transactions. A stable activity log is necessary for the durability or permanence of a committed transaction.

Figure 1 illustrates the execution flow of a two-phase commit atomic transaction. There are two synchronization points, *precommit* and *commit*, for each participating site. The coordinator begins a transaction by writing a *precommit* record into its activity log. The coordinator must be prepared to commit the transaction before writing the *precommit* record (i.e., updates have been flushed to stable log, resources are available to perform the commit, etc.). Writing the *pre-commit* record into the activity log tells the coordinator the status of the transaction if a failure occurs; the transaction has finished execution but is not yet committed. A vote request is then multicast to all participants. When a participant receives the vote request, it tests whether the transaction can be committed (the updates have been flushed to the activity log, serializability is ensured, resources are available, etc.). If the test is positive, the participant writes a *precommit* into the log and sends a YES reply to the coordinator. Otherwise, the

B.Tech. IV Sem J.C.S. Solved Papers

participant aborts the transaction and sends a NO reply to the coordinator.

If the coordinator is able to collect all YES replies within a time-out interval, it commits the transaction by writing a *commit* record in the log and multicasting a *commit* message to all participants. Otherwise, the coordinator aborts the transaction and multicasts an *abort* message. On receiving the *commit* message, each participant commits the transaction by writing a *commit* record into the activity log and releasing the transaction's resources. Finally, any response is returned to the coordinator. If the received message was an *abort*, the participant writes an *abort* record into the log, aborts the transaction, and releases the transaction's resources.

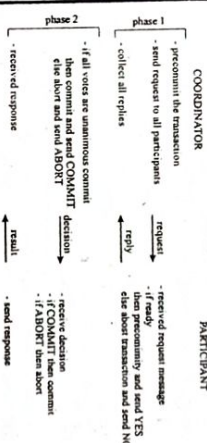


Fig. 1 : Two-phase commit Atomic transaction protocol

When used with an activity log, the two-phase commit protocol is highly resilient to processor failures. Figure 2 shows the time-lines of the protocol for the coordinator and a participant. Since writing *precommit* and *commit* to the activity log flushes all updates before these synchronization points, proper actions upon recovery from failures can rely on the replay of the log at least up to the synchronization points.

Thus recovery actions can be categorized into three types: failures before a *precommit*, failures after a *precommit* but before a *commit*, and failures after a *commit*. A processor (either the coordinator or a participant) can simply abort the transaction if it discovers from the log that the failure occurred before a *precommit*. This is equivalent to voting NO for the transaction. The coordinator can also abort the transaction if it crashes between *precommit* and *commit*, but it is more efficient to attempt to commit the transaction by remulticasting the request messages (if duplicates can be detected by the participants).

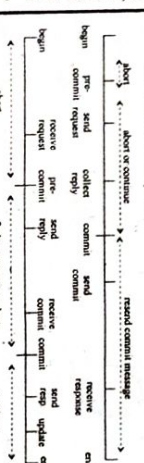


Fig. 2 : Failure and recovery action for the 2PC protocol. Similarly, it is safe to resend the commit messages if the coordinator crashes after writing the commit record to it

Distributed System

log. The recovery action is slightly more complicated if a participant crashes between *precommit* and *commit*. The recovering participant must determine whether the transaction was committed or aborted by contacting the coordinator or the other participants. Finally, if a participant recovers from a failure after a *commit* record has been written to the log, the participant simply makes the transaction's updates permanent. With stable storage, the recovery mechanisms ensure the durability of the commitments.

Prob.14 What are the basic difference between vector and matrix logical clock?

Sol. A logical clock is a mechanism for capturing chronological and causal relationships in a distributed system.

Vector clocks is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations. Just as in Lamport timestamps, interprocess messages contain the state of the sending process's logical clock. A vector clock of a system of N processes is an array/vector of N logical clocks, one clock per process, a local "smallest possible values" copy of the global clock-array is kept in each process, with the following rules for clock updates:

- Initially all clocks are zero.
- Each time a process experiences an internal event, it increments its own logical clock in the vector by one.
- Each time a process prepares to send a message, it increments its own logical clock in the vector by one and then sends its entire vector along with the message being sent.
- Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).
- A matrix clock is a mechanism for capturing chronological and causal relationships in a distributed system.
- Matrix clocks are a generalization of the notion of vector clocks. A matrix clock maintains a vector of the vector clocks for each communicating host.
- Every time a message is exchanged, the sending host sends not only what it knows about the global state of time, but also the state of time that it received from other hosts.
- This allows establishing a lower bound on what other hosts know and is useful in applications such as checkpointing and garbage collection.

Prob.15 Explain Two level concurrency of Process and Threads. [RTU 2013]

Sol. Process is a program in execution. It is a unit of execution. It is a sequence of instructions that are executed by the operating system. It is a unit of execution. It is a sequence of instructions that are executed by the operating system. It is a unit of execution. It is a sequence of instructions that are executed by the operating system.

Prob.18 Define the role of client and server stub procedures in RPC. Explain architecture of distributed event notification. [RTU 2011]

OR

What are the role of client and server stub procedures in RPC ? Explain with Diagram.

[Raj. Univ. 2008, 2007, 2005]

Sol. Remote Procedure Call : A remote procedure call is very similar to a remote method invocation in that a client program calls a procedure in another program running in a server process. Servers may be clients of other servers to allow chains of RPCs. A server process defines in its service interface the procedures that are available for calling remotely.

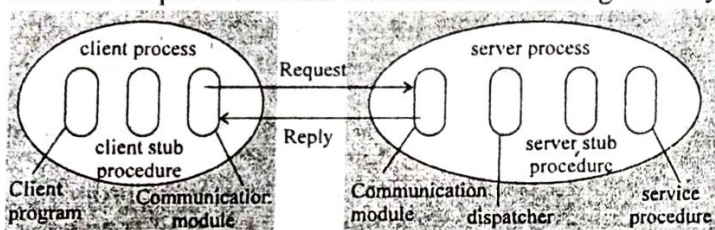


Fig. : Role of client and server stub procedures in RPC

The software that support RPC is shown in fig. In this there are no remote reference modules are required, since procedure call is not concerned with objects and object references. The client that accesses a service includes one stub procedure for each procedure in the service interface. The role of a stub procedure is similar to that of proxy. It behaves like a local procedure to the client but instead of executing the calls, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results. The server process contains a dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. A server stub procedure is like a skeleton method in that it unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the values for the reply message.

PART-C

Prob.19 What is synchronization and how it is required in concurrent process and programming? Explain. [R.T.U. 2019]

OR

Write short note on message passing.

[R.T.U. 2019]

OR

Describe the language mechanism for synchronization. [RTU 2016]

Sol. Synchronization in Distributed System : A distributed system is designed to realize some synchronized behavior, especially in real-time processing in areas like factories, aircraft, space vehicles, and military. Synchronization of individual clocks becomes very important in case of certain hard and risky real time applications like, where predictable performance is of major concern, one need to preserve a total logical or temporal scheduling of the tasks in the system.

Clock inaccuracies occur due to certain instabilities inherent in source oscillators and environmental conditions such as temperature, air circulation and mechanical stresses. The clocks in the different nodes need to be synchronized to limit the inaccuracies and hence implement the objectives of distributed system in an efficient manner. Hence, clock skew and drifts which forms the major source of clock inaccuracy needs to be monitored continuously. In certain applications it is not just enough to synchronize the various processes but also the various events that constitute them. This is called intraprocess synchronization. Intraprocess concurrency is captured by relation affects or causally affects. Bit matrix clocks and hierarchical clocks which evolved after the logical and vector clocks of Lamport capture affects relation between events of process. However both have a major disadvantage in terms of increased storage and communication overhead. Difference clocks captures intra-process and interprocess concurrency, at the same time with reduced storage and communication overheads.

Language Mechanism for Synchronization : With the background in basic process concepts, we now turn to the subject of language constructs for process interactions. A concurrent programming language allows the specification of concurrent processing and how concurrent processes synchronize and communicate with each other. It is natural to take an existing sequential language and add additional language facilities to support concurrent processing. This approach is easier for people to learn since there are fewer new concepts to acquire. A concurrent language extended from a sequential language adds additional constructs to provide:

- Specification of concurrent activities
- Synchronization of processes
- Interprocess communication
- Nondeterministic execution of processes.

Process synchronization has been widely studied for centralized operating systems. Process communication through message passing is a new issue when we consider distributed systems. This section summarizes some of the standard solutions for process.

1. Language Constructs

A common procedure-oriented language is generally defined by a complete specification of the syntactic structure and semantics of its major components. Typically these components can be categorized as follows:

- Program structure that specifies how programs and their subcomponents (procedures, blocks, statements, expression, terms, variables, constants, etc.) are composed. A sequential execution of program components is assumed unless explicitly altered by control statements.
- Data structures that are defined to represent objects in the program. Data type abstraction and its efficient implementation are primary goals.
- Control structures that regulate the flow of program execution. Most languages emphasize use of structured one-in-one-out control structures such as if-then-else, while-do, repeat-until, etc. Other implicit control structures include subroutine calls, returns, and exits.
- Procedures and system calls that activate special routines or system services. These calls imply change of execution flow and allow parameter passing.
- Input and output that receives input data and produces results of program execution. All programs contain at least some output operations. Input and output can be interpreted as a special case of message passing communication.
- Assignments that cause side effects for data objects. They are the primary operations that effect the state of program executions.

Table 1: Synchronization Mechanisms and Language Facilities

Synchronization Methods	Language Facilities
Shared-Variable Synchronization	shared variable and system call
semaphore	data type abstraction
monitor	control structure
conditional critical region	data type and control structure
Serializer	data type and program structure
path expression	data type and program structure
Message Passing Synchronization	input and output
communicating sequential processes	procedure call
remote procedure call	procedure call and communication
rendezvous	communication

2. Shared-Variable Synchronization

Shared-variable synchronization mechanisms were developed for synchronizing concurrent processes in a centralized operating system. As the name implies, process coordination is achieved by sharing variables. Interprocess communication uses neither message passing nor client/server communication. Although they are centralized approaches, they still have their place in the design of distributed operating systems. For example, the threads and tasks using distributed shared memory continue to use shared memory synchronization. It may also be desirable to simulate shared-variable synchronization by message passing.

(i) Semaphore: A Shared-variable and System Call Approach

Semaphores are a built-in system data type. A variable defined as a semaphore type is internally associated with a lock and a queue for process blocking purposes. Only two operations, P for acquiring the lock and V for releasing the lock, are allowed on semaphores for process synchronization. They are implemented as system calls to the operating system. The operating system maintains the indivisible property of the operations and is responsible for blocking and unblocking processes. The key characteristic of semaphores is that they provide only the most primitive locking mechanisms. Coordination of process activities to achieve correct synchronization is left entirely to the user processes. It is a shared memory concept and has virtually no transparency in hiding the synchronization details.

(ii) Monitor: A Data Type Abstraction Approach

A monitor is an object model concept and has a syntactic structure similar to a user-defined data type. A monitor consists of a declaration for its local variables, a set of allowable operations (or monitor procedures), and an initialization procedure that initializes the state of the monitor. The difference between a monitor and a regular user-defined data type is the semantic assumption that only one instantiation of a monitor object can be active at any time. This implicit assumption achieves the same mutual exclusion as a pair of P and V operations on a binary semaphore. However, the critical sections are fixed and predefined in the monitor procedures. Consequently, it is more structured than the semaphore approach. To coordinate activities once a process is in a monitor procedure, conditional variables with two standard operations, wait and signal, are used to suspend or resume execution of a process in the monitor. Because interleaving of monitor procedure activations is permitted, the implementation of monitors must ensure that no more than one process in the monitor is active at any time.

Since monitor procedures accept parameters, monitors provide process communication as well as synchronization. Clearly, shielding and shifting synchronization details from the user processes is a major step for the monitor approach toward providing transparency.

(iii) Conditional Critical Region: A Control Structure Approach

A Conditional Critical Region (CCR) is a structured version of the semaphore approach. Critical section codes are explicitly named and expressed by region-begin-end. Once in the critical region, a condition can be tested by the when predicate. If the condition is not met, the process is suspended and other processes are allowed to enter their critical regions.

(iv) Serializer: A Combined Data Abstraction and Control Structure Approach

In the discussion of the monitor synchronization method, we see that it is necessary to allow concurrency in the monitor and at the same time maintain the atomicity of the operation of a monitor procedure. A serializer is an extension of the monitor concept to achieve just that. Serializers have structures that are similar to those of monitors. Processes invoke serializer procedures in the same way that they invoke monitor procedure calls. Like monitors, exclusive access to the serializer is assumed. However, a serializer procedure consists of two types of regions, one that requires mutual exclusion and the other that allows concurrent processes to be active in the region; the latter is called a hollow region. This is accomplished by a new control structure, joincrowd-then-begin-end. When a process enters a hollow region, it releases the serializer and joins a crowd of concurrent processes. The blocking of processes by wait on condition variables in monitors is replaced by an enqueue operation in the serializer. The removal of a queued process when its waiting condition has been changed is done implicitly by the system rather than by using explicit signals as in the monitor approach. The use of a queue and implicit signaling greatly improves the clarity of programs.

(v) Path Expression: A Data Abstraction and Program Structure Approach

The path expression concept is significantly different from all the previously discussed synchronization methods. Like the monitors, it is an abstraction of data type. However, the procedures defined in the abstract data type contain no explicit reference to any synchronization primitives. Only the order of procedure execution must follow the constraints set by the path expressions. Path expressions are a high-level language specification that describes how operations defined for a shared object can be invoked to satisfy the synchronization requirements. For this reason we refer to it as a program structure approach since it resembles the formal description of a program. The path expression solution to the weak reader preference problem is extremely elegant:

path 1 (read, write) end

The constant 1 constrains the number of simultaneous activities in the parentheses to one and thus specifies the exclusion between reader and writer. The square brackets indicate that readers may be concurrent. Although elegant in

concept, path expressions must be sufficient in expressive power, and compilers must be available to translate path expressions into implementable synchronization primitives. Many extensions to the path expression concept have been explored to increase its capability in specifying concurrency and synchronization requirements. The most notable example is the inclusion of predicates in path expressions for condition coordination.

3. Message Passing Synchronization

An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution), unless the client sends an asynchronous request to the server, such as an XMLHttpRequest. There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols.

Without shared memory, message passing is the only means for communication in distributed systems. Message passing is also a form of implicit synchronization since messages can be received only after they have been sent. For most applications, it is common to assume that receive is blocking, and send may be either blocking or nonblocking. We will call nonblocking send, blocking receive asynchronous message passing and blocking send, blocking receive synchronous message passing. This section describes how synchronization can be accomplished using the two different assumptions of message passing. Concepts relevant to message passing such as Communicating Sequential Processes (CSP), remote procedure calls, and rendezvous are discussed.

(i) Asynchronous Message Passing

Although there is no sharing of variables in message passing, the communication channel is shared. Therefore, the blocking receive operation from a message channel is equivalent to a P operation on a semaphore, and the nonblocking send is analogous to a V operation. Asynchronous message passing is simply an extension of the semaphore concept to distributed systems. Note that asynchronous send operations assume that the channel has an unbounded buffer and so perhaps cannot be fully implemented. Asynchronous message passing synchronization is as useful as semaphores if communication channels can be specified in the language and supported by the operating system. Figure 1 demonstrates the use of asynchronous message passing for mutual exclusion. The channel server represents the operating system support of logical channels. It creates the logical channel solely for synchronization purposes and initializes the channel to contain one message. The message content in the channel is

Synchronous message passing assumes blocking send and blocking receive. This is necessary when there is no buffering of messages in the communication channel. A sender must wait for the corresponding receive to occur, and a receiver must wait for the corresponding send. In other words, whichever comes first must wait for the other, and the waiting is symmetrical. This mechanism allows two processes with a matching pair of send and receive to join and exchange data at a synchronization point and to continue their separate execution thereafter. This type of synchronization is called a rendezvous between send and receive. Rendezvous is a useful concept in computer systems, as well as in real life. Outside the stadium before the start of a football game, it is quite common to see people with their hands up either holding tickets to be sold or sticking up fingers to indicate how many tickets they need. Rendezvous of sellers and buyers take place asynchronously and symmetrically.

Output Approach

Communicating Sequential Processes (CSP) was one of the first languages to address the synchronization problem in distributed systems. It uses input/output rendezvous to achieve synchronous message passing synchronization. Input/output is a form of message communication. A sender process *P* can issue an output command, *Q*!exp, to a receiver process *Q* with a corresponding input command, *P*?var. The rendezvous of the input and output commands is connected through explicit naming of each other's process names. The effect is equivalent to a remote assignment statement that assigns the value of exp in one process to the variable var of another remote process. The message exchanged between the input and the output processes is synchronous and thus achieves implicit synchronization between two processes.

Prob.20 Describe the design issues and implementation of RMI
[RTU 2016]

OR
Explain design and implementation issues in Remote Method Invocation (RMI).
[RTU 2018]

Discuss the design and implementation issues in Remote Method Invocation.
[R.T.U. 2017]

Discuss designing issues for RMI.
[Raj. Univ. 2007, 2006, 2004]

- Although local invocations are executed exactly once, this cannot always be the case for remote method invocations.

- The level of transparency that is desirable for RMI. The processes that host remote objects as servers and the processes that host their invokers as clients. Servers can also be clients.

1. RMI Invocation Semantics : Request-reply protocols such as the Do operation that can be implemented in different ways to provide different delivery guarantees.

The main choices are :

Retry Request Message : Whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

Duplicate Filtering : When retransmissions are used, rather to filter out duplicate requests at the Server.

Retransmission of Results : Whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

Table : Invocation Semantics

Fault tolerance measures		Re-execute	Invocation semantics
Retransmit request message	Duplicate filtering	procedure or retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

The above table shows the various types of invocation semantics.

Combinations of these choices lead to a variety of possible semantics for the reliability of invocations as seen by the invoker.

Table shows the choices of interest, with corresponding names for the invocation semantics that they produce. Note that for local method invocations, the semantics are called exactly once, meaning that every method is executed exactly once.

The invocation semantics are described below :

May invocation semantics: With invocation semantics

the invoker cannot tell whether remote method has been executed once or not at all. May be semantics arises when none of the fault tolerance measures is applied. This can suffer following types of failure :

object fails.

If the result message has not been received after a time-out and there are no retries, it is uncertain whether the method has been executed. If the invocation message was lost then

has occurred. If the invocation message was lost, the method will not have been executed. On the other hand, if the message was received but not processed, the method may have been executed and the result message lost. A crash failure may occur before or after the method is executed. Moreover, in an asynchronous system, the results of executing the method may arrive after the time-out. Many of the semantics useful only for applications in which occasional failed invocations are acceptable.

Atleast-once invocation

once invocation semantics, the invoker receives either a result, in which case the invoker knows that the method was executed at least once, or an exception informing it that no result was received. At least–once invocation semantics can be achieved by the retransmission of request message. At least once invocation semantics can suffer from the following types of failure :

- Crash failures when the server containing the remote object fails

- **Arbitrary failures**
In cases when the invocation message is retransmitted, the remote object may receive it and execute the method more than once, possibly causing wrong values to be stored or returned.

At-most-once invocation semantics : With at-most-once invocation semantics, the invoker receives either a result, or a fault. The fault is returned if the method was executed in which case the invoker knows that the method was executed exactly once, or an exception informing it that no result was received, in which case the method will have been executed either once or not at all. At-most-once invocation semantics can be achieved by using all of the fault tolerance measures.

The next designing issue in RMI is described below:

2. Transparency : The originators of RPC, aimed to make remote procedure calls as possible with no distinction in syntax between a local and a remote procedure call. All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call. Although request masses are retransmitted after a time out, this is transparent to the caller—to make the semantics of remote procedure call like that of local procedure calls. This notion of transparency has been extended to apply to distributed objects, but it involves hiding not only marshalling and message passing but also the task of locating and contacting a remote object. As an example, Java RMI makes

remote method invocations very like local ones by allowing them to use the same syntax. However, remote invocations are more vulnerable to failure than local ones, since they involve a network, another computer and another process.

Whichever of the above invocation semantics is chosen, there is always the chance that no result will be received and in the case of failure, it is impossible to distinguish between the failure of the network and of the remote server process. This failure of the network and of the remote server process to recover from such several orders of magnitude greater than that of a local one. This suggests that programs that make use of remote invocations need to be able to take this factor into account, perhaps by minimizing remote interactions.

Implementation of RMI : The RMI model used in

Java for distributed processing is based fundamentally on the client-server model. Recall that in this model, a number of client applications (e.g., web browsers) connect to a central server (e.g., a web server) to carry out processing. In Java RMI, client objects interact with server objects to carry out processing. What Java RMI adds to the client-server model is a specific, concrete mechanism for Java objects (and thus, the underlying Java interpreters, i.e., Java virtual machines) to communicate. This communication is carried out via remote procedure calls which are simply a mechanism by which methods of remote Java objects can be invoked by other local Java objects. Thus, using RMI requires the following steps:

1. Define one or more Java classes that provide functionality for the server in the distributed application.

2. Register one or more methods of these classes as being remotely accessible.
3. Define one or more Java classes that provide

functionality for the clients in the distribution application. The clients can make use of the remotely accessible methods in the servers by looking up the available methods and invoking them, using Java RMI, in their own code.

The client first contacts an RMI registry to look up the name of a service. This directs the client to the location of the service. The client can then dynamically load the requested object from a web server; this happens automatically.

We illustrate the use of the RML model with parts of a small example.

```
// Calculate the i-th Fibonacci number
public long fib ( long i )
throws RemoteException
{
    double res;
```



```
res = java.lang.Math.pow(phi,1)/java.lang.Math.sqrt(5);
return java.lang.Math.round(res);
}
```

For efficiency, the method uses a closed formula for calculating Fibonacci numbers. The exact approach used for this calculation is unimportant.

We now implement the constructor and main method for the server. The constructor is straightforward, as all we must do is ensure that the constructor can throw a RemoteException (which can be caught).

```
public FibonacciService() throws
RemoteException
{
    super();
}
```

Finally, we must implement the main method. Most of the complexity of the RMI application is here. This method is responsible for creating an instance of the Fibonacci server, registering the service with the RMI registry, and attaching a security manager to the server. For a simple application like this, we strictly need not include a security manager. It is generally good practice to think about security throughout the development process when building distributed systems.

```
public static void main (String args[]) throws Exception
{
    // Attach a security manager.
    if ( System.getSecurityManager() == null )
        System.setSecurityManager(new RMISecurityManager());
    // Create instance of the Fibonacci server.
    FibonacciService f = new FibonacciService();
    // Bind it to the RMI registry.
    Naming.bind("FibonacciService", f);
    System.out.println("Fibonacci Service registered.");
}
```

Implementing the client: In order to provide a useful, working application, we need to provide a client that makes use of the Fibonacci server. In order to make use of a server that is accessible via RMI, we first call the registry to obtain the right remote object, then invoke its methods. This is quite straightforward. For our specific example, we first attach a security manager, then call the RMI registry to access the Fibonacci service, as follows.

```
// Attach a security manager
if ( System.getSecurityManager() == null )
{
    System.setSecurityManager(new RMISecurityManager());
}
// Find the Fibonacci service in the registry
FibonacciService f = (FibonacciService) Naming.lookup(
    "rmi:// " + args[0] + "/" + "FibonacciService");
```

The Naming lookup method call asks the registry to find a specific service. The service is identified by using an RMI URL. This RMI URL is another naming convention, used to identify the location of an RMI service. An RMI URL contains the host name on which a service is located, and the logical name of the service. This returns a FibonacciService instance, which can then be used just as if it was a local object. So for example, to find the twelfth Fibonacci number, we could write

```
System.out.println("Number is " + f.fib(12));
```

All that remains is to compile the application and execute it. To do this, a registry must be started (how to do this depends on your operating system, for example, using the call startmregistry under Windows). When starting the RMI client, you can run the client locally, or from a remote machine.

Prob 21) What is RPC in distributed systems? Explain the client-server architecture. Describe the role of client stub and server stub while making an RPC call. [RTU 2016]

OR

Explain the concept of RPC in detail. [R.T.U. 2018]

OR

Where do you need RPC? Explain with suitable example. [R.T.U. 2017]

OR

Where do you need RPC? Explain with a suitable example. [RTU 2014]

OR

Explain Client Server Communication model on RPC and message passing? [RTU 2013]

OR

What is RPC? Explain its protocol and working. [RTU 2012]

Sol. RPC is an interprocess communication technique that allows client and server software to communicate. It allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote.

RPC Working: An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller

waits for a response to be returned from the remote procedure. Figure (a) shows the flow of activity that takes place during an RPC call between two networked systems.

The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

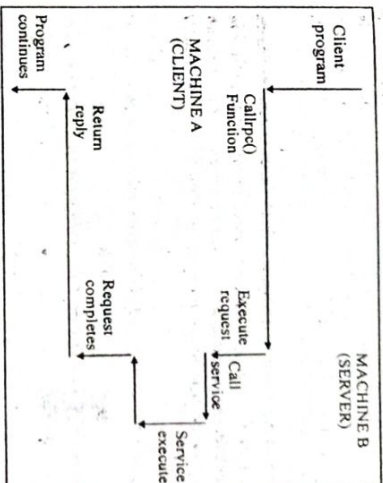


Fig. (a) : Message Passing

Message Passing : Refer to Prob. 19.

Sequence of Events During an RPC

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the server stub.
5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.
6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

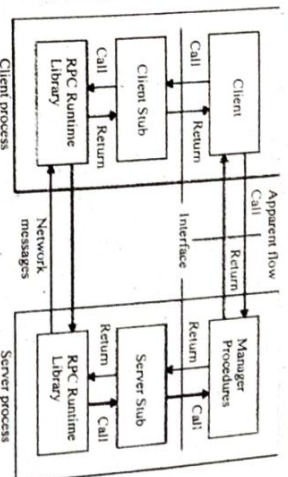


Fig. (b) : Remote Procedure Call Flow

RPC Message Protocol

The Remote Procedure Call (RPC) message protocol consists of two distinct structures: the call message and the reply message. A client makes a remote procedure call to a network server and receives a reply containing the results of the procedure's execution. By providing a unique specification for the procedure's execution, RPC can match a reply message to each call (or request) message.

RPC Protocol Requirements

The RPC message protocol requires:

- Unique specification of a procedure to call
- Matching of response messages to request messages
- Authentication of caller to service and service to caller

To help reduce network administration and eliminate protocol roll-over errors, implementation bugs, and user errors, features that detect the following conditions are useful:

- RPC protocol mismatches
- Remote program protocol version mismatches
- Protocol errors (such as misspecification of a procedure's parameters)
- Reasons why remote authentication failed
- Any other reasons why the desired procedure was not called

RPC Messages

The initial structure of an RPC message is as follows:

```
struct rpe_msg {
    unsigned int xid;
    union switch (enum msg_type mtype) {
        case CALL:
            call_body body;
        case REPLY:
            reply_body body;
    } body;
};
```


DS.36

All RPC call and reply messages start with a transaction identifier, *xid*, which is followed by a two-armed discriminated union. The union's discriminant is **msg_type**, which switches to one of the following message types : CALL or REPLY. The **msg_type** has the following enumeration :

```
enum msg_type {
    CALL    = 0,
    REPLY   = 1
};
```

The *xid* parameter is used by clients matching a reply message to a call message or by servers detecting retransmissions. The server side does not treat the *xid* parameter as a sequence number.

The initial structure of an RPC message is followed by the body of the message. The body of a call message has one form. The body of a reply message, however, takes one of two forms, depending on whether a call is accepted or rejected by the server.

• RPC Call Message

Each remote procedure call message contains the following unsigned integer fields to uniquely identify the remote procedure :

- Program number
- Program version number
- Procedure number

The body of an RPC call message takes the following form :

```
struct call_body {
    rpcvers_t rpcvers;
    rpcprog_t prog;
    rpcvers_t vers;
    rpcproc_t proc;
    opaque_auth cred;
    opaque_auth verf;
    1 parameter
    2 parameter ...
};
```


Prob.4 What are the rules of committing nested transactions?

Sol. Rules of Committing Nested Transactions

- A transaction may commit or abort only after its child transactions have completed.
- When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort.
- When a parent aborts, all of its transactions aborted.

Prob.5 What is file replication?

Sol. In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits – its enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service.

PART-B

Prob.6 What is transaction service and concurrency control? Explain. [R.T.U. 2019]

OR

Discuss transaction services and concurrency control in detail. [RTU 2012]

Sol. A transaction is the basic logical unit of execution in an information system. A transaction is a sequence of operations that must be executed as a whole.

Requirements for Database Consistency

The simultaneous execution of many different application programs must be such that each transaction does not interfere with another transaction. The concurrent execution of

Distributed System

transactions must be such that each transaction appears to execute in isolation.

Desirable Properties of Transactions (ACID)

Atomicity: A transaction is an atomic unit of processing and it is either performed entirely or not at all.

Consistency Preservation: A transaction's correct execution must take the database from one correct state to another.

Isolation: The updates of a transaction must not be made visible to other transactions until it is committed (solves the temporary update problem).

Durability or Permanency: If a transaction changes the database and is committed, the changes must never be lost because of subsequent failure.

Serializability: Transactions are considered serializable if the effect of running them in an interleaved fashion is equivalent to running them serially in some order. A mechanism controls concurrency among database transactions through the use of serial ordering relations. The ordering relations are computed dynamically in response to patterns of use. An embodiment of the present invention serializes a transaction that accesses a resource before a transaction that modifies the resource, even if the accessor starts after the modifier starts or commits after the modifier commits.

A method of concurrency control for a database transaction in a distributed database system stores an intended use of a database system resource by the database transaction in a serialization graph.

A serialization ordering is asserted between the database transaction and other database transactions based on the intended use of the database system resource by the database transaction.

The Need for Concurrency Control: The concurrent execution of transactions may lead, if uncontrolled, to problems such as an inconsistent database. Concurrency control techniques are used to ensure that multiple transactions submitted by various users do not interfere with one another in a way that produces incorrect results.

Prob.7 What is distributed process implementation and also explain static process scheduling with communication. [R.T.U. 2017]

OR

What is static process scheduling? Write the advantages of static process scheduling. [RTU 2016]

OR

What is static process scheduling?

Give the advantages of static process scheduling.

Sol. Distributed Logical Model

Processes are distributed between the processes at

(i) Interpretation

(ii) Invocation

the RP

Application

request messages application

(i) Remote

request

(ii) Remote

to be executed

(iii) Process

process

the execution

Mobile Agent

travel across a network

provide agent

migrate from

into multiple

and to return to

Unlike

procedures of

executable code

systems, information

Mobile

next. When a

state and transp

execution from

Static Process

In static

processors is

Information reg

resources is ass

is always execu

assigned, that is

non-preemptive

methods is to m

concurrent progr

delays. With this

attempt to:

- (i) Predict the program execution behavior at compile time (that is, estimate the process or task, execution times, and communication delays).
- (ii) Perform a partitioning of smaller tasks into coarser-grain processes in an attempt to reduce the communication costs.
- (iii) Allocate processes to processors.

One of the most critical shortcomings of static scheduling is that, in general, generating optimal schedules is an NP-complete problem. It is only possible to generate optimal solutions in restricted cases (for example, when the execution time of all of the tasks is the same and only two processors are used).

Advantages of static scheduling :

- The major advantage of static scheduling methods is that all the overhead of the scheduling process is incurred at compile time, resulting in a more efficient execution time environment compared to dynamic scheduling methods.
- Guaranteed scheduling-on time
- An activity is always allowed to finish - no concurrent threads, no critical regions
- Easy to calculate worst-case execution time for the activities
- Short guaranteed response times
- Inflexible, sensitive to changes
- Inefficient, can not use free time in the time slots.

Disadvantages of static scheduling : Refer to Prob. 1.

Prob. 8 Define time stamp ordering in transaction and explain with suitable example. [RTU 2016, 2010]

OR

Explain the method of Timestamp ordering.

[RTU 2010]

Sol. Timestamp Ordering : In concurrency control schemes based on timestamp ordering, each operation in a transaction is validated when it is carried out. If the operation cannot be validated, the transaction is aborted immediately and can then be restarted by the client. Each transaction is assigned a unique timestamp value when it starts. The timestamp defines its position in the time sequence of transactions. Requests from transactions can be totally ordered according to their timestamps. The basic timestamp ordering rule is based on operation conflicts and is very simple :

A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.

This rule assumes that there is only one version of each object and restricts access to one transaction at a time. If each transaction has its own tentative version of each object it accesses, then multiple concurrent transactions can access the same object. The timestamp ordering rule is refined to ensure that each transaction accesses a consistent set of versions of the objects. It must also ensure that the tentative versions of each object are committed in the order determined by the timestamps of the transactions that made them. This is achieved by transactions waiting, when necessary, for earlier transactions to complete their writes. The write operations may be performed after the *close transaction* operation has returned, without making the client wait. But the client must wait when *read* operations need to wait for earlier transactions to finish.

Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	write	read	T_i must not write an object that has been read by any T_j where $T_j > T_i$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	write	write	T_c must not write an object that has been written by any T_j where $T_j > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	read	write	T_c must not read an object that has been written by any T_j where $T_j > T_c$ this requires that $T_c >$ write timestamp of the committed object.

This cannot lead to deadlock, since transactions only wait for earlier ones (and no cycle could occur in the wait-for graph).

Timestamps may be assigned from the server's clock or, a 'pseudo-time' may be based on a counter that is incremented whenever a timestamp value is issued.

As usual, the write operations are recorded in tentative versions of objects and are invisible to other transactions until a *close transaction* request is issued and the transaction is committed. Every object has a write timestamp and a set of tentative versions, each of which has a write timestamp associated with it; and a set of read timestamps. The write timestamp of the (committed) object is earlier than that of any of its tentative versions, and the set of read timestamps can be represented by its maximum member. Whenever a transaction's write operation on an object is accepted, the server creates a new tentative version of the object with write timestamp set to the transaction timestamp. A transaction's read operation is directed to the version with the maximum write timestamp less than the transaction timestamp. Whenever a transaction's read operation on an object is accepted, the timestamp of the transaction is added to its set of read timestamps. When a transaction is committed

the values of the tentative versions become the values of the objects, and the timestamps of the tentative versions become the timestamps of the corresponding objects.

In timestamp ordering, each request by a transaction for a read or write operation on an object is checked to see whether it conforms to the operation conflict rules. A request by the current transaction T_c can conflict with previous operations done by other transactions, T_i , whose timestamps indicate that they should be later than T_c these rules are shown in Table, in which $T_i > T_c$ means T_i is later than T_c and $T_i < T_c$ means T_i is earlier than T_c .

Prob. 9 Explain the working of Bit Torrent file system with a neat diagram. [RTU 2015]

Sol. A typically, internet access is asymmetrical, supporting greater download speeds than upload speeds, limiting the bandwidth of each download, and sometimes enforcing bandwidth caps and periods where systems are not accessible. This creates inefficiency when many people want to obtain the same set of files from a single source; the source must always be online and must have massive outbound bandwidth. The BitTorrent protocol addresses this problem by decentralizing the distribution by creating a peer-to-peer network.

Each file to be distributed is divided into small information chunks called pieces. Downloading peers achieve rapid download speeds by requesting multiple pieces from different computers simultaneously in the network. The sub-network of peers thus formed is called as 'swarm'. Once obtained, these pieces are immediately made available for download by others in the swarm. In this way, the burden on the network is spread among the downloaders, rather than concentrating at a central distribution hub or cluster. As long as all the pieces are available, peers (downloaders and uploaders) can come and go; no one peer needs to have all the chunks, or to even stay connected to the swarm in order for distribution to continue among the other peers.

A small torrent file is created to represent a file or folder to be shared. It contains metadata about files and folders to be distributed and a list of the network locations of trackers, which are servers that help participants in the system find each other.

The torrent file acts as the key to initiating downloading of the actual content. The user first downloads the torrent file. User then opens that file in a BitTorrent client, which automates the rest of the process.

In order to learn the Internet locations of peers which may be sharing pieces, the client connects to the trackers

Distributed System

UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes.

Prob.15 Compare the forward and backward validation with starvation.

Sol. Comparison of forward and backward validation :

Forward validation allows flexibility in the resolution of conflicts, whereas backward validation allows only one choice to abort the transaction being validated. In general, the read sets of transactions are much larger than the write sets. Therefore, backward validation compares a possibly large read set against the old write sets, whereas forward validation checks a small write set against the read sets of active transactions. The backward validation has the overhead of storing old write sets until they are no longer needed. On the other hand, forward validation has to allow for new transactions starting during the validation process.

Starvation : When a transaction is aborted, it will normally be restarted by the client program. But in schemes that rely on aborting and restarting transactions, there is no guarantee that a particular transaction will ever pass the validation checks, for it may come into conflict with other transactions for the use of objects each time it is restarted. The prevention of a transaction ever being able to commit is called starvation.

Occurrences of starvation are likely to be rare, but a server that uses optimistic concurrency control must ensure that a client does not have its transaction aborted repeatedly. Kung and Robinson suggest that this could be done if the server detects a transaction that has been aborted several times. They suggest that when the server detects such a transaction it should be given exclusive access by the use of a critical section protected by a semaphore.

PART-C

Prob.16 Explain Dynamic Load Sharing and balancing and its requirement in distributed process scheduling with justification. [R.T.U. 2019]

OR

Why do you need dynamic load sharing and balancing? Explain. [RTU 2014]

OR

Explain dynamic load sharing and balancing methods in detail along with appropriate example. [RTU 2018]

DS.49

Sol. It is desirable in a distributed system to have the system load balanced evenly among the nodes so that the mean job response time is minimized. In this section, we present a dynamic load balancing mechanism (DLB). It adopts a centralized approach and is network topology independent. The DLB mechanism employs a set of thresholds which are automatically adjusted as the system load changes. It also provides a simple mechanism for the system to switch between periodic and instantaneous load balancing policies with ease. The performance of the proposed algorithm is evaluated by intensive simulations for various parameters. The simulation results show that the mean job response time in a system implementing DLB algorithm is significantly lower than the same system without load balancings. Furthermore, compared with a previously proposed algorithm, DLB algorithm demonstrates improved performance, especially when the system is heavily loaded and the load is unevenly distributed.

Keywords: Distributed computing, load balancing, centralized scheduling, homogeneous distributed system, distributed operating system.

Parallel and distributed systems are considered by many experts to be the future for scientific and engineering computing. A distributed system is a collection of autonomous processors interconnected by a communication network. The processors communicate with each other through the network to share resources. The main design goal of a distributed system is short mean job response time. A load balancing mechanism is a software approach to redistribute system-wide workloads among the computers in the system in order to reduce the mean job response time. In order to achieve this goal, a load balancing mechanism should have the following features: (1) It generates as little traffic overhead as possible while keeping load information updated promptly. (2) It creates a low overhead on the CPU for running the load balancing algorithm. (3) It achieves fairness: the load of the lightest-loaded node and the heaviest-loaded node should be balanced first. (4) Load balancing is done in reasonably short time intervals.

Load balancing mechanisms can be classified as centralized or decentralized, dynamic or static, periodic or instantaneous (non-periodic), and with thresholds or without thresholds.

In a centralized load balancing system, the global load information is collected at a single node, called the central scheduler. The other nodes, called local nodes, send their load-update messages to the central scheduler. All load balancing decisions are made at the central scheduler based on the collected messages. A main problem in centralized load balancing mechanisms is the relatively low reliability. Failure of the central scheduler means the failure of the entire load

DS.50

balancing mechanism. In a decentralized load balancing system, each node broadcasts its load information periodically to the other nodes to update their locally maintained load tables. Every node performs its node selection action. This can be done by having every node keep track of the global system load status continually. The major drawback with such an approach is that every node, including busy ones, must keep track of incoming update messages and generate update messages of its own. Centralized algorithms generate less overhead on the system than the distributed ones, therefore can support a larger system. Theimer concluded that the main trade-off between the two approaches is that a centralized architecture can be scaled to a much greater degree and can more easily monitor global system statistics, while a decentralized architecture is simpler to implement.

Static mechanisms assume knowledge of the main system parameters and dispatch jobs according to rules that are set a priori and are not affected by the current state of the system. Static load balancing is simple and effective when the workload can be sufficiently well characterized beforehand, but it cannot adjust to the fluctuations in system load. In contrast, dynamic load balancing tries to redistribute the system load dynamically as jobs arrive. Previous studies have shown that dynamic load balancing mechanisms generally have a greater performance improvement than static mechanisms. It has received increasing attention from the research community.

Load balancing algorithms can also be classified into periodic or instantaneous depending on how frequently they are implemented. Periodic mechanism starts load balancing in predetermined time intervals. However, instantaneous mechanism starts load balancing whenever the load status of some nodes changes.

A load balancing mechanism could implement a threshold policy or without any thresholds. By applying threshold policy, workloads are usually divided into three categories: light-load, heavy-load, and normal-load. Individual nodes report to the central scheduler only when their load status changes. In contrast, in a load balancing mechanism that does not have a threshold policy, whenever a job arrives or departs at a node, the node reports to the central scheduler.

The load balancing mechanism proposed in this section is a centralized and dynamic one. It uses a threshold policy and provides both periodic and instantaneous load balancings.

Prob.17 Explain Data and file Replication in Distributed File Systems. [R.T.U. 2019]

OR

Write short note on data and file replication. [RTU 2018]

Sol. Data Replication : Data replication is the process of copying data from one location to another. The technology helps an organization possess up-to-date copies of its data in the event of a disaster.

Replication can take place over a storage area network, local area network or local wide area network, as well as to the cloud. For disaster recovery (DR) purposes, replication typically occurs between a primary storage location and a secondary offsite location.

Approaches to Data Replication : There are four places where replication can happen: in the host, hypervisor, storage array or network. Array-based replication once was the dominant method, but the others have gained in popularity.

Host-based replication uses servers to copy data from one site to another, using software on application servers. It is usually file-based and asynchronous. Host-based replication software includes capacities such as deduplication, compression, encryption and throttling.

Hypervisor-based replication is a type of host-based replication that replicates entire virtual machines from one host server or host cluster to another. Because it is specifically designed for VMs, hypervisor replication makes it easy to fail over to the replicate if the primary copy of the VM is lost. And it can run on servers that do not natively support replication. All host-based replication uses CPU resources, which may impact server performance.

Array-based replication allows compatible storage arrays to use built-in software to automatically copy data between arrays. Array-based replication is more resilient and requires little cross-departmental coordination when deployed. But it is limited to homogeneous storage environments, as it requires similar source and target arrays.

Network-based replication requires an extra switch or appliance between storage arrays and servers. Network-based replication typically takes place in heterogeneous storage environments — it works with any array and supports any host platform. There are fewer network-based replication products on the market compared to array- and host-based offerings.

Synchronous Vs. Asynchronous Data Replication : Data replication can be synchronous or asynchronous, depending on when it takes place.

Synchronous Replication : Synchronous replication takes place in real time, and is preferred for applications with low recovery time objectives that can't lose data. It's primarily used with high-end transactional applications that require instantaneous failover in the event of a failure. This replication approach is more expensive and creates latency that slows the primary application.

Synchronous replication is supported by array-based and most network-based replication products, but rarely in host-based ones.

Asynchronous Replication : Asynchronous replication is time-delayed. It is designed to work over distances and requires less bandwidth.

This replication is intended for businesses that can withstand lengthier recovery point objectives. Because there is a delay in the copy time, the two data copies may not always be identical. Asynchronous replication is supported by array-, network- and host-based replication products.

File-based Replication : File-based replication conducts data replication at the logical level (i.e.: individual data files) rather than at the storage block level. There are many different ways of performing this, which almost exclusively rely on software.

Capture with a Kernel Driver : A kernel driver (specifically a filter driver) can be used to intercept calls to the filesystem functions, capturing any activity as it occurs. This utilises the same type of technology that real-time active virus checkers employ. At this level, logical file operations are captured like file open, write, delete, etc. The kernel driver transmits these commands to another process, generally over a network to a different machine, which will mimic the operations of the source machine. Like block-level storage replication, the file-level replication allows both synchronous and asynchronous modes. In synchronous mode, write operations on the source machine are held and not allowed to occur until the destination machine has acknowledged the successful replication. Synchronous mode is less common with file replication products although a few solutions exist.

File-level replication solutions allow for informed decisions about replication based on the location and type of the file. For example, temporary files or parts of a filesystem that hold no business value could be excluded. The data transmitted can also be more granular; if an application writes 100 bytes, only the 100 bytes are transmitted instead of a complete disk block (generally 4096 bytes). This substantially reduces the amount of data sent from the source machine and the storage burden on the destination machine.

Drawbacks of this software-only solution include the requirement for implementation and maintenance on the operating system level, and an increased burden on the machine's processing power.

File System Journal Replication : Similarly to database transaction logs, many file systems have the ability to journal their activity. The journal can be sent to another machine, either periodically or in real time by streaming. On the replica

side, the journal can be used to play back file system modifications.

One of the notable implementations is Microsoft's System Center Data Protection Manager (DPM), released in 2005, which performs periodic updates but does not offer real-time replication.

Prob.18 Give the brief description of DFS Design and Implementation. [RTU 2013]

OR

What is distributed file system? Explain the concept of transaction service and concurrency control in detail. [RTU 2018]

OR

What do you mean by distributed file system? Explain its design and implementation, [RTU 2012]

Sol. Distributed file system or network file system is any file system that allows access to files from multiple hosts sharing via a computer network. This makes it possible for multiple users on multiple machines to share files and storage resources.

The client nodes do not have direct access to the underlying block storage but interact over the network using a protocol. This makes it possible to restrict access to the file system depending on access lists or capabilities on both the servers and the clients, depending on how the protocol is designed. Distributed file systems may include facilities for transparent replication and fault tolerance. That is, when a limited number of nodes in a file system go offline, the system continues to work without any data loss.

Design Issues

Naming and name resolution

Terminology

- **Name :** each object in a file system (file, directory) has a unique name
- **Name resolution :** mapping a name to an object or multiple objects (replication)
- **Name space :** collection of names with or without same resolution mechanism

Approaches to naming files in a distributed system

(a) Concatenate name of host to names of files on that host

- **Advantage :** unique file names, simple resolution
- **Disadvantages:**
 - Conflicts with network transparency
 - Moving file to another host requires changing its name and the applications using it

Uniquifier :

This is a unique number to ensure that the same vnode IDs are not reused.

Each server maintains a copy of a database that maps a volume number to its server. If the client request is incorrect (because a volume moved to a different server), the server forwards the request. This provides AFS with migration transparency: volumes may be moved between servers without disrupting access.

Concept of Transaction Service and Concurrency Control : Refer to Prob.6.

Prob.19 Write short notes on followings :

(a) General parallel file system and window's file system [R.T.U. 2017]

(b) Andrew and coda file systems [R.T.U. 2017]

OR

When do you prefer coda file system? Justify your answer. [RTU 2015]

OR

Describe the coda file system. [RTU 2014, 2012]

OR

How does AFS works in Coda file System? [RTU 2013]

OR

Describe Andrew file system. [RTU 2014]

OR

(c) Sun network file system. [R.T.U. 2017, 2012]

Sol.(a) General Parallel File System (GPFS) : It is a high-performance clustered file system developed by IBM. It can be deployed in shared-disk or shared-nothing distributed parallel modes. It is used by many of the world's largest commercial companies, as well as some of the supercomputers on the Top 500 List. For example, GPFS was the file system of the ASC Purple Supercomputer which was composed of more than 12,000 processors and has 2 petabytes of total disk storage spanning more than 11,000 disks.

In common with typical cluster file systems, GPFS provides concurrent high-speed file access to applications executing on multiple nodes of clusters. It can be used with AIX 5L clusters, Linux clusters, on Microsoft Windows Server, or a heterogeneous cluster of AIX, Linux and Windows nodes. In addition to providing file system storage capabilities, GPFS provides tools for management and administration of the GPFS cluster and allows for shared access to file systems from remote GPFS clusters.

Window's File System : Window's file system is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of

information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names is called a "file system".

File systems can be used on numerous different types of storage devices that use different kinds of media. The most common storage device in use today is a hard disk drive. Other kinds of media that are used include flash memory, magnetic tapes, and optical discs. In some cases, such as with tmpfs, the computer's main memory (random-access memory, RAM) is used to create a temporary file system for short-term use.

Sol. (b) Andrew File Systems

The Andrew File System (AFS) is a distributed network file system that enables files from any AFS machine across the country to be accessed as easily as files stored locally.

AFS is composed of cells, with each cell representing an independently administered portion of file space. Cells connect to form one enormous UNIX file system under the root/afs directory. PSC organizes and maintains the disk space associated with the cell psc.edu.

Features

AFS has several benefits over traditional networked file systems, particularly in the areas of security and scalability. It is not uncommon for enterprise AFS cells to exceed 25,000 clients. AFS uses Kerberos for authentication and implements access control lists on directories for users and groups. Each client caches files on the local file system for increased speed on subsequent requests for the same file. This also allows limited file system access in the event of a server crash or a network outage.

- Read and write operations on an open file are directed only to the locally cached copy. When a modified file is closed, the changed portions are copied back to the file server. Cache consistency is maintained by callback mechanism. When a file is cached, the server makes a note of this and promises to inform the client if the file is updated by someone else. Callbacks are discarded and must be re-established after any client, server or network failure, including a time-out. Re-establishing a callback involves a status check and does not require re-reading the file itself.
- A consequence of the file locking strategy is that AFS does not support large shared databases or record updating within files shared between client systems. This was a deliberate design decision based on the perceived needs of the university computing environment. It leads, for

example, to the use of a single file per message in the original email system for the Andrew Project, the Andrew Message System rather than a single file per mailbox.

A significant feature of AFS is the volume, a tree of files, sub-directories and AFS mountpoints (links to other AFS volumes). Volumes are created by administrators, and linked at a specific named path in an AFS cell. Once created, users of the file system may create directories and files as usual without concern for the physical location of the volume. A volume may have a quota assigned to it in order to limit the amount of space consumed. As needed, AFS administrators can move that volume to another server and disk location without the need to notify users; indeed the operation can occur while files in that volume are being used.

AFS volumes can be replicated to read-only cloned copies. When accessing files in a read-only volume, a client system will retrieve data from a particular read-only copy. If at some point that copy becomes 'unavailable', clients will look for any of the remaining copies. Again, users of that data are unaware of the location of the read-only copy; administrators can create and relocate such copies as needed. The AFS command suite guarantees that all read-only volumes contain exact copies of the original read-write volume at the time the read-only copy was created.

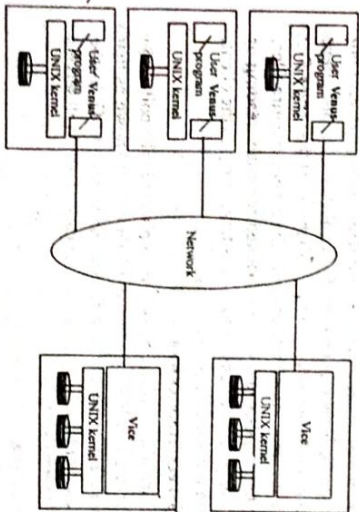


Fig. 1 : AFS architecture : Venus, network and Vice

The file name space on an Andrew workstation is partitioned into a shared and local name space. The shared name space (usually mounted as /afs on the Unix file system) is identical on all workstations. The local name space is unique to each workstation. It only contains temporary files needed for workstation initialization and symbolic links to files in the shared name space.

The Andrew File System heavily influenced Version 4 of Sun Microsystems' popular Network File System (NFS).

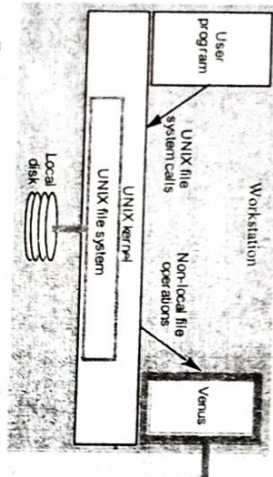


Fig. 2 : AFS system call intercept handling by Venus

User process	UNIX kernel	Venus	Vice
Open file	Open file in local cache. If not found, request for the file from the server.	Check file in local cache. If not found, request for the file from the server.	Transfer a copy of the file from the server to the local cache.
Read file	Read file from local cache.	Read file from local cache.	Read file from local cache.
Write file	Write file to local cache.	Write file to local cache.	Write file to local cache.
Close file	Close file in local cache.	Close file in local cache.	Close file in local cache.

Fig. 3 : Implementation of file system calls-callbacks and callback promises

Coda File System : Coda was designed to be a scalable, secure, and highly available distributed file system. An important goal was to achieve a high degree of naming and location transparency so that the system would appear to its users very similar to a pure local file system. By also taking high availability into account, the designers of Coda have also tried to reach a high degree of failure transparency. Coda is a descendant of version 2 of the Andrew File System (AFS).

Coda follows the same organization as AFS. Every Virtue workstation hosts a user-level process called Venus, whose role is similar to that of an NFS client. A Venus process is responsible for providing access to the files that are maintained by the Vice file servers. In Coda, Venus is

also responsible for allowing the client to continue operation even if access to the file servers is (temporarily) impossible. This additional role is a major difference with the approach followed in NFS.

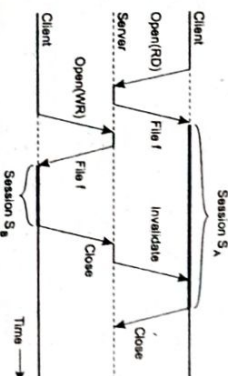
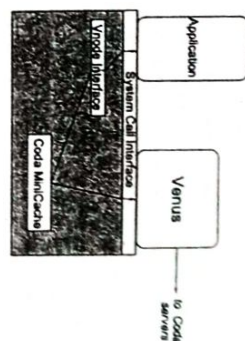


Fig. 4

Communication

Interprocess communication in Coda is performed using RPCs. However, the RPC2 system for Coda is much more sophisticated than traditional RPC systems. RPC2 offers reliable RPCs on top of the (unreliable) UDP protocol. Each time a remote procedure is called, the RPC2 client code starts a new thread that sends an invocation request to the server and subsequently blocks until it receives an answer. As request processing may take an arbitrary time to complete, the server regularly sends back messages to the client to let it know it is still working on the request. If the server dies, sooner or later this thread will notice that the messages have ceased and report back failure to the calling application.

A side effect is a mechanism by which the client and server can communicate using an application-specific protocol. Consider, for example, a client opening a file at a video server. What is needed in this case is that the client and server set up a continuous data stream with an isochronous transmission mode.

Processes

Coda maintains a clear distinction between client and server processes. Clients are represented by Venus processes; servers appear as Vice processes. Both type of

processes are internally organized as a collection of concurrent threads. Threads in Coda are non-preemptive and operate entirely in user space. To account for continuous operation in the face of blocking I/O requests, a separate thread is used to handle all I/O operations, which it implements using low-level asynchronous I/O operations of the underlying operating system. This thread effectively emulates synchronous I/O without blocking an entire process.

Naming

Files are grouped into units referred to as volumes. A volume is similar to a UNIX disk partition (i.e., an actual file system), but generally has a much smaller granularity. It corresponds to a partial subtree in the shared name space as maintained by the Vice servers. Usually a volume corresponds to a collection of files associated with a user. Examples of volumes include collections of shared binary or source files, and so on. Like disk partitions, volumes can be mounted.

File Identifiers

Collection of shared files may be replicated and distributed across multiple Vice servers, it becomes important to uniquely identify each file in such a way that it can be tracked to its physical location, while at the same time maintaining replication and location transparency.

Client Caching

Client-side caching is crucial to the operation of Coda. First caching is done to achieve scalability. Second, caching provides a higher degree of fault tolerance as the client becomes less dependent on the availability of the server. For these two reasons, clients in Coda always cache entire files. In other words, when a file is opened for either reading or writing, an entire copy of the file is transferred to the client, where it is subsequently cached.

Server Replication

Coda allows file servers to be replicated. The collection of servers that have a copy of a volume, are known as that volume's Volume Storage Group, or simply VSG. In the presence of failures, a client may not have access to all servers in a volume's VSG.

Sol(c) Sun Network File System

Network File System (NFS) is a distributed file system protocol originally developed by Sun Microsystems allowing a user on a client computer to access files over a network in a manner similar to how local storage is accessed. The Network File System is an open standard defined in RFCs, allowing anyone to implement the protocol.

Design Goals : NFS was designed to simplify the sharing of filesystem resources in a network of non-homogeneous machines. Our goal was to provide a way of making remote

PREVIOUS YEARS QUESTIONS

PART-A

Prob.1 What do you mean by thrashing?

Sol. It is a computer activity that makes little or no progress, usually memory or other resources have become exhausted or too limited to perform needed operations.

Prob.2 What is deadlock condition?

Sol. A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.

Prob.3 Write the requirements for pervasive applications.

Sol. Three requirements for pervasive applications

- **Embrace contextual changes :** A device must be continuously be aware of the fact that its environment may change all the time.
- **Encourage ad hoc composition:** Many devices in pervasive systems will be used in very different ways by different users – make it easy to configure the suite of applications running on a device.
- **Recognize sharing as the default:** Devices generally join the system in order to access (and possibly provide) information.

Prob.4 What is full form NUMA? Where it is used?

Sol. NUMA stands for Non-Uniform Memory Architecture is a computer memory design used in multiprocessors where the memory access time depends on the memory location relative to a processor.

PART-B

Prob.5 What is Distributed Shared memory? Explain its implementation in distributed systems. [R.T.U. 2019]

OR

Explain different DSM implementation approaches.

[RTU 2012]

OR

Explain DSM systems along with their implementations and applications. [RTU 2018]

OR

What is the implementation of DSM system? [R.T.U. 2017]

Sol. Distributed Shared Memory (DSM) : In Computer architecture DSM is a form of memory architecture where the (physically separate) memories can be addressed as one (logically shared) address space. Here, the term **shared** does not mean that there is a single centralized memory but **shared** essentially means that the address space is shared (same physical address on two processors refers to the same location in memory).

With DSM, local as well as remote memories can be accessed in a uniform manner, with the location of the shared region transparent to the application program. DSM is a major area of interest as it provides a better price/performance ratio, improved speed and reliability but problems arise due to communication delays and consistency. DSM appears as shared memory to the applications programmer, but relies on message passing between independent CPUs to access the global virtual address space.

DSM Implementation Issues

1. How to keep track of the location of remote data?
2. How to overcome the communication delays and high overhead associated with the execution of communication protocols in distributed systems when accessing remote data?

3. How to make shared data concurrently accessible at several nodes in order to improve system performance?

DSM Implementation Algorithms :

(i) Client-server Algorithm

Client request data and server sends the requested data to client.

(ii) Migration Algorithm

The data is shipped to the location of the data request, allowing subsequent accesses to be performed locally

(iii) Read-replication Algorithm

Replicate the data blocks and allow multiple nodes to have read access or one node to have read-write access.

Application of Distributed Shared Memory (DSM) :

- Scales well with a large number of nodes.
- Message passing is hidden.
- Can handle complex and large databases without replication or sending the data to processes.
- Generally cheaper than using a multiprocessor system.
- Provides large virtual memory space.
- Programs are more portable due to common programming interfaces.
- Shield programmers from sending or receiving primitives.

Prob.6 Explain failures in a distributed system in distributed computation. [R.T.U. 2019]

OR

Explain the failures in a distributed system.

[RTU 2016]

OR

What are the different types of failures in distributed systems? Explain. [RTU 2014]

Sol. There are different types of failure across the distributed system and few of them are given below :

Crash Failures: Crash failures are caused across the server of a typical distributed system and if these failures are occurred operations of the server are halt for some time. Operating system failures are the best examples for this case and the corresponding fault tolerant systems are developed with respect to these affects.

Timing Failures: Timing failures are caused across the server of a distributed system. The usual behavior of these timing failures would be like that the server response time towards the client requests would be more than the expected range. Control flow out of the responses may be caused due to these timing failures and the corresponding clients may give up as they can't wait for the required response from the server and thus the server operations are failed due to this.

Omission Failures: Omission failures are caused across the server due to lack of reply or response from the server across the distributed systems. There are different issues raised due to these omission failures and the key among them are server not listening or a typical buffer overflow error across the servers of the distributed systems.

Byzantine Failures: Byzantine failures are also known as arbitrary failures and these failures are caused across the server of the distributed systems. These failures cause the server to behave arbitrary in nature and the server responds in an arbitrary fashion at arbitrary times across the distributed systems. Output from the server would be inappropriate and there could be chances of the malicious events and duplicate messages from the server side and the clients get arbitrary and unwanted duplicate updates from the server due to these failures.

Prob.7 Explain distributed deadlock handling in distributed computation. [R.T.U. 2019]

Sol. Distributed Deadlock Handling : In Distributed Deadlock Handling, a deadlock is detected in the following way. The process (or thread) which needs a resource to get itself running sends a probe message to the process which is currently holding the required resource. A probe message contains 3 fields : the process ID that is blocked, the process ID that is sending the request, and the destination process ID. The blocked ID is same as the sender process ID. If the receiver process is not waiting for its own execution, it can release the resource and the first process can use the resource. But if the receiver process itself is waiting on a resource or a set of resources, it will update the sender process ID and the destination and forwards the probe to the processes holding the required resource set. This process continues till the processes are waiting for resources and if the original process receives the probe and checks its own ID in the blocked field, it will know that a deadlock exists in the system. It can choose to kill itself or some other mechanism can be used to kill another process.

In Centralized Deadlock handling, there is a central coordinator which detects and handles a deadlock condition by looking for cycles in the WFG (Wait-for-Graph). Every machine on the distributed system maintains a graph containing its resources and processes and the central coordinator maintains graph for the whole distributed system. This graph is a union of all the individual machine graphs. The individual machines sends messages to the coordinator telling about the release of some resource or requesting for a resource.

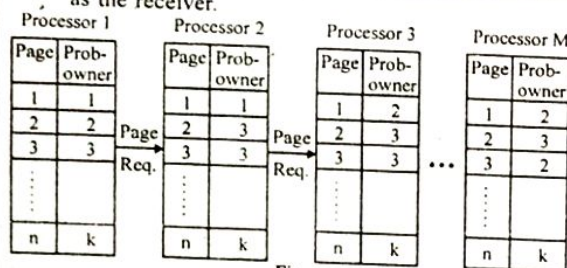
A cycle in the global graph means there is a deadlock condition and the coordinator has to kill a process.

Sol. Dynamic distributed manger algorithm :

In this management algorithm every host keeps track of the page ownership in its local page table. Page table has a column probowner (probable owner) whose value can either be the true owner or a probable one (i.e., it is used as a hint). Initially set to a default value. On a page fault, request is sent to i (assume). If i is the owner, steps proceed as in the case of central manager. If not, request is forwarded to probowner of p at i . This is done until the actual owner is reached.

Probowner is updated on receipt of:

1. Invalidation requests, ownership relinquishing messages.
2. **Receiving or forwarding of a page :** For writes, the receiver is the owner. Forwarder updates the owner as the receiver.



At most $(N-1)$ messages needed to locate the owner. As hints are updated as a side effect, the average number of messages should be lower.

Double fault : Consider a processor p doing a read first and then a write. p needs to get the page twice.

One solution : Use sequence numbers along with page transfers. p can send its page sequence number to the owner. Owner can compare the numbers and decide whether a transfer is needed or not. Still checking with owner is needed, only transfer of whole page is avoided.

Thrashing : Thrashing is computer activity that makes little or no progress, usually because memory or other resources have become exhausted or too limited to perform needed operations. When this happens, a pattern typically develops in which a request is made of the operating system by a process or program, the operating system tries to find resources by taking them from some other process, which in turn makes new requests that can't be satisfied. In a virtual storage system (an operating system that manages its logical storage or memory in units called pages), thrashing is a condition in which excessive paging operations are taking place.

A system that is thrashing can be perceived as either a very slow system or one that has come to a halt.

Prob.10 Describe mechanism for deadlock detection in distributed system. [R.T.U. 2017, 2011]

Sol. Distributed Deadlock Detection :

Deadlock occurring at a single site is called as local deadlock whereas those involving transactions executing at multiple sites is called a global deadlock. In distributed deadlock detection, global deadlocks are detected by sending inter-site deadlock detection messages. Detection schemes for global deadlocks are classified into two categories depending upon the type of graph they construct, which is either an actual graph or a condensed graph.

Actual graph detection schemes are based upon transmission of detection messages conveying strings of transactions of an arbitrary length in which each transaction (global/local) waits for the completion of the immediately following transaction in the string, the first transaction being the global one being waited by some other site and the last transaction being the global transaction that is waiting for either a response or a new request from another site. The deadlock detection message (\Rightarrow string of transactions) is sent to the site for which the last transaction in the string is waiting and is used to construct an actual deadlock detection graph at the receiving site of the message. The messages thus transmitted can be reduced into half by assigning priorities to transactions and transmitting only those messages where the priority of the first transaction in the string is higher than the priority of the last transaction in the string.

In the condensed graph detection scheme the deadlock messages contain only two transactions say (T_i, T_j) where transaction T_i transitively waits for the completion of transaction T_j . The message is sent either to the originating site of transaction T_i (backward transmission) or to the originating site of transaction T_j (forward transmission). In order to prevent the large number of message transfers probe is sent when a transaction with a higher priority transitively waits for another transaction with a lower priority which avoid the transmission of backward messages. An interesting feature in the modified probe scheme is that once a probe is received it is stored and forwarded until no more paths are found for delivering the probe or a compensating message for the probe called antiprobe is received. However it is not free from drawbacks in the sense that they send messages to transaction managers and also to resource managers (resulting in the transmission of a lot of deadlock detection and resolution messages even when only local transactions are concerned in a site) and they treat local and global transactions equally.

Various approaches for deadlock detection in distributed systems

(i) Path-pushing Algorithms

The basic idea underlying this class of algorithms is to build some simplified form of global WFG at each site. For this purpose each site sends its local WFG to a number of

neighboring sites every time a deadlock computation is performed. After the local data structure of each site is updated, this updated WFG is then passed along and the procedure is repeated until some site has sufficiently complete picture of the global situation to announce deadlock or to establish that no deadlocks are present.

(ii) Edge-chasing Algorithms

The presence of a cycle in a distributed graph structure can be verified by propagating special messages called probes along the edges of the graph. Probes are assumed to be distinct from resource request and grant messages. When the initiator of such a probe computation receives a matching probe, it knows that it is in cycle in the graph. A nice feature of this approach is that executing processes can simply discard any probes they receive. Blocked processes propagate the probe along their outgoing edges.

Prob.11 Explain distributed shared memory system in distributed environment with suitable diagram. [RTU 2016]
OR

What is Distributed Shared Memory? Explain any memory management algorithm in detail.

Sol. Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying run-time system ensures transparently that processes executing at different computers observe the updates made by one another. It is as though the processes access a single shared memory, but in fact the physical memory is distributed.

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client server systems, where clients normally view server-held resources as abstract data and access them by request (for reasons of modularity and protection). However, servers can provide DSM that is shared between clients. For example, memory-mapped files that are shared and for which some degree of consistency is maintained are a form of DSM. (Mapped files were introduced with the MULTICS operating system.)

Message passing cannot be avoided altogether in a distributed system in the absence of physically shared memory, the DSM run-time support has to send updates in messages between computers. DSM systems manage replicated data :

sequentially over a network) and some problems associated with each algorithm.

DS.69

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Non token based			
(i) Centralized	3	2	Coordinator crash
(ii) Distributed	$2(n-1)$	$2(n-1)$	Crash of any process
Token ring	1 to ∞	0 to $n-1$	Lock token, process crash

Fig. : A comparison of three mutual exclusion algorithms

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request, a grant to enter, and a release to exit. The distributed algorithm requires $n > 1$ request messages, one to each of the other processes and an additional $n > 1$ grant messages, for a total of $2(n > 1)$. (We assume that only point-to-point communication channels are used.)

With the token ring algorithm, the number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry all varies for the three algorithms. When critical regions are short and rarely used, the dominant factor or in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. In Fig. above we show the former case. It takes only two message times to enter a critical region in the centralized case, but $2(n > 1)$ message times in the distributed case, assuming that messages are sent one after the other. For the token ring, the time varies from 0 (token just arrived) to $n > 1$ (token just departed).

PART-C

Prob.17 Describe the concept of deadlocks in distributed systems. Explain token based algorithm and non-token based algorithm in mutual exclusion. [RTU 2018]

Sol. Distributed Deadlock Introduction : A deadlock is a condition in a system where a set of processes (or threads) have requests for resources that can never be satisfied. Essentially, a process cannot proceed because it needs to

DS.70

obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, Coffman defined four conditions have to be met for a deadlock to occur in a system:

- Mutual Exclusion** : A resource can be held by at most one process.
- Hold and Wait** : Processes that already hold resources can wait for another resource.
- Non-preemption** : A resource, once granted, cannot be taken away.
- Circular Wait** : Two or more processes are waiting for resources held by one of the other processes.

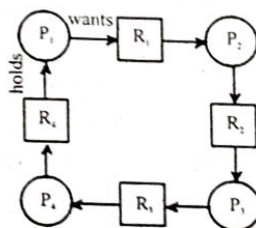


Fig. 1 : Deadlock

A directed graph model used to record the resource allocation state of a system. This state consists of n processes, $P_1 \dots P_n$, and m resources, $R_1 \dots R_m$. In such a graph:

$P_i \rightarrow R_j$ means that resource R_j is allocated to process P_i .

$P_i \leftarrow R_j$ means that resource R_j is requested by process P_i .

Deadlock is present when the graph has a directed cycles. An example is shown in figure 1. Such a graph is called a Wait-For Graph (WFG).

Deadlock in Distributed Systems

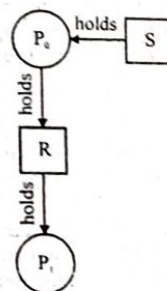


Fig. 2 : Resource graph on A

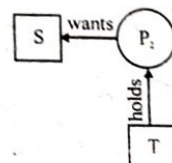


Fig. 3 : Resource graph on B

B.Tech. (VI Sem.) C.S. Solved Papers

The same conditions for deadlock in uniprocessors apply to distributed systems. Unfortunately, as in many other aspects of distributed systems, they are harder to detect, avoid, and prevent. Four strategies can be used to handle deadlock.

- Ignorance** : Ignore the problem, assume that a deadlock will never occur. This is a surprisingly common approach.
- Detection** : Let a deadlock occur, detect it, and then deal with it by aborting and later restarting a process that causes deadlock.
- Prevention** : Make a deadlock impossible by granting requests so that one of the necessary conditions for deadlock does not hold.
- Avoidance** : Choose resource allocation carefully so that deadlock will not occur. Resource requests can be honored as long as the system remains in a safe (non-deadlock) state after resources are allocated.

The last of these, deadlock avoidance through resource allocation is difficult and requires the ability to predict precisely the resources that will be needed and the times that they will be needed. This is difficult and not practical in real systems. The first of these is trivially simple but, of course, ineffective for actually doing anything about deadlock conditions. We will focus on the middle two approaches.

In a conventional system, the operating system is the component that is responsible for resource allocation and is the ideal entity to detect deadlock. Deadlock can be resolved by killing a process. This, of course, is not a good thing for the process. However, if processes are transactional in nature, then aborting the transaction is an anticipated operation. Transactions are designed to withstand being aborted and, as such, it is perfectly reasonable to abort one or more transactions to break a deadlock. The transaction can be restarted later at a time when, we hope, it will not create another deadlock.

Prob.18 Explain distributed mutual exclusion with its classification. [RTU 2016]

OR

Explain the distributed mutual exclusion. [RTU 2018]

Sol. Distributed Mutual Exclusion : A distributed system is a collection of autonomous computers connected via a communication network. There is no common memory and processes communicate through message passing. One of the most important purposes of the distributed systems is to provide an efficient and convenient environment for sharing of resources. They also provide computational speedup and better reliability. In a system consisting of a number of

PART-A

Prob.1 What do you mean by CORBA?

Sol. CORBA stands for Common Object Request Broker Architecture. It is a standard defined by object management group that enables software components written in multiple computer languages and running on multiple computers to work together.

Prob.2 Define ORB.

Sol. The software that implements the CORBA specifications is called the ORB (Object Request Broker).

Prob.3 What are object adapters?

Sol. An object adapter is the primary way that an object implementation accesses services provided by the ORB. Such services include object reference generation and interpretation, method invocation.

Prob.4 Write the use of interface repository.

Sol. Interface Repository : The IR provides another way to specify the interfaces to objects. Interfaces can be added to the interface repository service. Using the IR, a client should be able to locate an object that is unknown at compile time, find information about its interface, then build a request to be forwarded through the ORB.

Prob.5 Write two techniques used for dealing failures in a distributed system.

Sol. Two Techniques used for Dealing Failures in a Distributed System :

1. **Detecting Failure :** Some failures can be detected. For eg : Checksums can be used to detect corrupted data in a message or a file.
2. **Masking Failures :** Some failures that have been detected can be hidden or made less serve.

PART-B

Prob.6 Explain the concept of replicated data management and various issues associated with Replicated data management in detail. [R.T.U. 2019]

OR

State and explain various issues of replicated data management. [R.T.U. 2018]

OR

Discriminate passive replication and active replication. [R.T.U. 2017]

OR

Write short note on the Replicated data management. [RTU 2016]

OR

What is Replication? Explain in detail. [RTU 2013]

OR

Explain replication of data in detail. Also give its classification. [RTU 2012]

Sol. Replication Resolution : Once a conflict is detected, steps must be taken to resolve the conflict. Resolution involves choosing new contents for the item that was updated concurrently and generating a new version that supersedes previous known versions. The item's new contents could be taken from one of the conflicting versions or produced by

merging the conflicting versions in some manner. In any case, when all conflicts have been resolved, the system should be left with a single latest version of each data item. Issues in conflict resolution include how, when, and where to resolve conflicts as well as how to ensure that all replicas eventually agree on the conflict resolution and thereby converge to a consistent state.

How are Conflicts Resolved : Distributed systems have generally used one of three main approaches for resolving conflicts. Each of these approaches may be preferred in some environments and may better meet the needs of particular applications. Therefore, some systems provide all three as options.

One, the system can rely on users to choose the "winner" from among two or more conflicting versions of an item, which is referred to as manual conflict resolution.

Two, a system can have a built-in conflict resolution policy or allow system administrators to select from a well-defined set of policies that are applied to each detected conflict. When a conflict is detected, the winning version is chosen without human involvement based on the configured policy.

Three, a system can allow applications to register conflict resolvers that are automatically invoked when a conflict is detected. Such resolvers are software routines that are presented with the conflicting versions, allowed to take arbitrary action, and expected to return new contents for the item in conflict.

Where are Conflicts Resolved : Regardless of the approach taken to resolve conflicting items, a question remains about when and where conflict resolution takes place. In other words, which devices in a system are responsible for performing manual or automatic conflict resolution. There are two basic alternatives: the resolve-everywhere model and resolve-anywhere model.

In the resolve-everywhere scheme, conflicting versions of an item fully propagate to all replicas, and each device independently detects conflicts and resolves them locally. Resolutions do not propagate to other devices; they simply affect a device's local replica. This is the approach that has been taken by most replication systems.

In the resolve-anywhere scheme, when a device detects a conflict and resolves it locally, the device also propagates its resolution to other replicas. Conflict resolution produces new updates that are sent via the normal replication protocol and overwrite the conflicting versions. This works well in systems with manual resolution since a human need only resolve a conflict on one device as well as in systems that combine manual and automatic conflict resolution.

Replication : Replication in computing involves sharing information so as to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.

The replication of data in a distributed system offers the following potential benefits :

1. Increased Availability

One of the important advantages of replication is that it marks and tolerates failures in the network gracefully. The system remains operational and available to the users despite failures. By replicating critical data on servers with independent failure modes, the probability that one copy of the data will be accessible increases. Therefore alternate copies of a replicated data can be used when the primary copy is unavailable.

2. Increased Reliability

Many applications require extremely high reliability of their data stored in files. Replication is very advantageous for such applications because it allows the existence of multiple copies of their files. Due to the presence of redundant information in the system, recovery from catastrophic failures become possible.

3. Improved Response Time

Replication also helps in improving response time because it enables data to be accessed either locally or from a node to which access time is lower than the primary copy access time. The access time differential may arise either because of network topology or because of uneven loading of nodes.

4. Reduced Network Traffic

If a file's replica is available with a file server that resides on a client's node, the client's access requests can be serviced locally, resulting in reduced network traffic.

5. Improved System Throughput

Replication also enables several client's requests for access to the same file to be serviced in parallel by different servers, resulting in improved system throughput.

6. Better Scalability

As the number of users of a shared file grows, having all access requests for the file serviced by a single file server can result in poor performance due to overloading of the file server. By replicating the file on multiple servers, the same requests can now be serviced more efficiently by multiple servers due to workload distributed. This results in better scalability.

7. Autonomus Operation

In a distributed system that provides file replication as a service to their clients, all files required a client for operation during a limited time period may be replicated on the file server residing at the client's node. This will facilitate temporary autonomous operation of client machines. A distributed system having this feature can support detachable, portable machines.

Active and Passive Replication in Distributed Systems :

In active replication each client request is processed by all the servers. This requires that the process hosted by the servers is deterministic. Deterministic means that, given the same initial state and a request sequence, all processes will produce the same response sequence and end up in the same final state. In order to make all the servers receive the same sequence of operations, an atomic broadcast protocol must be used. An atomic broadcast protocol guarantees that either all the servers receive a message or none, plus that they all receive messages in the same order. The big disadvantage for active replication is that in practice most of the real world servers are non deterministic. Still active replication is the preferable choice when dealing with real time systems that require quick response even under the presence of faults or with systems that must handle Byzantine faults.

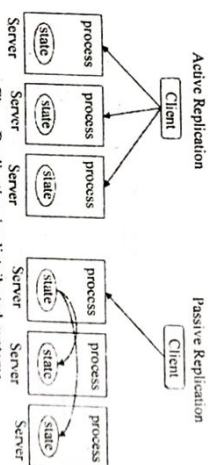


Fig. : Replication in distributed systems

In **passive replication** there is only one server (called primary) that processes client requests. After processing a request, the primary server updates the state on the other (backup) servers and sends back the response to the client. If the primary server fails, one of the backup servers takes its place. Passive replication may be used even for non deterministic processes. The disadvantage of passive replication compared to active is that in case of failure the response is delayed.

Prob 7 Explain CORBA RMI and services in detail.

OR [R.T.U. 2019]

Explain CORBA Services.

OR [R.T.U. 2018]

Write short note on CORBA RMI.

OR [RTU 2017, 2016, 2014]

Explain the case study of CORBA RMI.

OR [RTU 2010]

Write short notes on CORBA Services.

OR [RTU 2015]

Sol. CORBA Services : Many type of services described by CORBA are :

(1) **Naming Service :** It allows names to be bound to the remote object references of CORBA object within naming contexts.

(2) **Event Service :** Defines interfaces allowing objects of interest called suppliers to communicate notifications to subscribers.

(3) **Notification Service :** Defined as data structures, event suppliers are provided with a means of discovering the events, the consumers and interested in.

(4) Security Service

(a) **Authentication :** Of principles, generating gedetious for principles.

(b) **Access control** can be applied to CORBA objects when they receive remote method invocations.

(c) **Auditing** by servers of remote method invocations.

(5) **Trading Service :** In contrast to the Naming Service which allows CORBA objects to be located by name, the Trading Service allows them to be located by attribute - it is directory service. Its database contains a mapping from service types and their associated attributes onto remote object references of CORBA objects. The service type is a name, and each attribute is a name-value pair. Clients make queries by specifying the type of service required, together with other arguments specifying constraints on the values of attributes, and preferences for the order in which to receive matching offers. Trading servers can form federations in which they not only use their own databases but also perform queries on behalf of one another's clients.

(6) **Transaction Service and Concurrency Control Service :** The object transaction service allows distributed CORBA objects to participate in either flat or nested transactions. The client specifies a transaction as a sequence of RMI calls, which are introduced by begin and terminated by commit or rollback (abort). The ORB attaches a transaction identifier to each remote invocation and deals with begin, commit and rollback (abort) requests. Clients can also suspend and resume transactions. The transaction service carries out a two phase commit protocol. The concurrency control service uses locks to apply concurrency control to the access of CORBA objects. It may be used from within transactions or independently.

(7) **Persistent Object Service :** Persistent objects can be stored in a passive form in a persistent object store while they are not in use and activated when they are needed. Although ORBs activate CORBA objects with persistent objects references, getting their implementations from the implementation repository, they are not responsible for saving and restoring the state of CORBA objects. Instead, each CORBA object is responsible for saving and restoring its

own state, for example by making use of files. The CORBA Persistent Object Service (POS) is intended to be suitable for use as a persistent object store for CORBA objects. The specification defines interfaces for its components. The POS can be implemented in a variety of ways, for example, by using files or by using a relational or object-oriented database system.

The architecture of the POS allows for a set of data stores to be available - each persistent object has a persistent identifier which includes the identity of its data store and an object number within that store. Either the client or the CORBA object may decide when to save its state, by sending a request to the POS which forwards it to the appropriate data store. To allow its clients to control its persistence, a CORBA object inherits an interface that includes operations for saving, restoring or deleting it.

The implementation of a persistent CORBA object must choose a protocol for communicating with a data store. For example, it might transfer its data via a stream. The POS specification offers a selection of alternative protocols for this task, providing a range of functionalities. For example, the direct access protocol allows CORBA objects to access the attributes of persistent objects in the data store. In this case, the implementor of the CORBA object uses an IDL-like data definition language to define the attributes in its persistent state.

CORBA RMI

Programming in a multi-language RMI system such as CORBA RMI requires more of the programmer than programming in a single-language RMI system such as Java RMI. The following new concepts need to be learned:

- The object model offered by CORBA;
- The interface definition language and its mapping onto the implementation language.

In particular, the programmer defines remote interfaces for the remote objects and then uses an interface compiler to produce the corresponding proxies and skeletons. But in CORBA, proxies are generated in the client language and skeletons in the server language. We will use the simple whiteboard example to illustrate how to write an IDL specification and to build server and client programs.

CORBA's Object Model : The CORBA object is a model but clients are not necessarily objects - a client can be any program that sends request messages to remote objects and receives replies. The term CORBA object is used to refer to remote objects. Thus, a CORBA object implements an IDL interface, has a remote object reference and is able to respond to invocations of methods in its IDL interface. A

CORBA object can be implemented by a language that is not object-oriented, for example without the concept of class. Since implementation languages will have different notions of class or even none at all, the class concept does not exist in CORBA. Therefore classes cannot be defined in CORBA IDL, which means that instances of classes cannot be passed as arguments. However, data structures of various types and arbitrary complexity can be passed as arguments.

CORBA IDL : A CORBA IDL interface specifies a name and a set of methods that clients can request. Two interfaces named *Shape* and *Shape List*, which are IDL versions of the interfaces. These are preceded by definitions of two structs, which are used as parameter types in defining the methods. Note in particular that *Graphical Object* is defined as a struct, whereas it was a class in the Java RMI example. A component whose type is a struct has a set of fields containing values of various types like the instance variables of an object, but it has no methods.

Prob.8 Write short note on Atomic multicast.

[RTU 2017, 2016, 2014]

Sol. Atomic Multicast

Atomic group multicast is implemented over the reliable group multicast protocol. Like reliable multicast, the atomic multicast protocol provides an interface for members to multicast messages to the group, and it delivers a sequence of events to each member, where each event is a group view or a message. The atomic multicast protocol delivers the same sequence of events to each correct member that is not removed from the group.

Our atomic multicast protocol is similar to those of the Amoeba and Isis systems, which are tolerant to only benign failures. In these systems, there is a designated member of each group view, called the sequencer, that determines the order in which atomic multicast to that view are delivered, using one of two methods. In the first method, called "PB" the member initiating an atomic multicast does so by sending the message to the sequencer. Upon receiving the message, the sequencer forwards it to the group using a FIFO multicast, and all members deliver atomic multicast messages in the order they are received from the sequencer. In the second method, called "BB", the member initiating the atomic multicast FIFO-multicasts the message, say *m*, to the group. Upon receiving *m*, the sequencer FIFO-multicasts a second message naming *m* as the next atomic multicast message to be delivered. With either method, atomic multicasts are delivered in the same order at all group members because their order is dictated by the sequencer. Techniques to handle

DS-92

undermining one of the advantages claimed for replication.

A common scheme for dealing with this problem (not the only one possible) is the so-called primary copy scheme, which works as follows:

- One copy of each replicated object is designated as the primary copy. The remainder are all secondary copies.
- Primary copies of different objects are at different sites (so this is a distributed scheme once again).
- Update operations are deemed to be logically complete as soon as the primary copy has been updated. The site holding that copy is then responsible for propagating the update to the secondary copies at some subsequent time.

Note: That "subsequent time" must be prior to COMMIT. However, if the ACID properties of the transaction are to be preserved.

Of course, this scheme leads to several additional problems of its own.

Note too that it does represent a violation of the local autonomy objective, because a transaction might now fail because a remote (primary) copy of some object is unavailable even if a local copy is available.

Now, we have said that the ACID requirements of transaction processing imply that all update propagation must be completed before the relevant transaction can complete ("synchronous replication"). However, several commercial products support a less ambitious form of replication, in which update propagation is done at some later time (possibly at some user-specified time), not necessarily within the scope of the relevant transaction ("asynchronous replication"). Indeed, the term replication has unfortunately been more or less usurped by those products, with the result that—in the commercial marketplace, at least—it is almost always taken to imply that update propagation is delayed past COMMIT. One problem with this delayed propagation approach is that the database can no longer be guaranteed to be consistent at all times; indeed, the user might not even know whether it is consistent or not.

PART-C

Prob. 10 Explain the concept of faults, failure and recovery. [R.T.U. 2019, 2016]

OR

Explain the concepts of fault, failure and recovery in distributed systems. [R.T.U. 2018]

B.Tech. (VI Sem.) C.S. Solved Papers

OR

Explain Fault, Errors and Failure in Distributed System in brief. [R.T.U. 2013]

OR

Write short note on Failure and Recovery in DS. [R.T.U. 2017]

Sol. Fault Failure : A failure occurs when an actual running system deviates from this specified behavior. The cause of a failure is called an error. An error represents an invalid system state, one that is not allowed by the system behavior specification. The error itself is the result of a defect in the system or fault. In other words, a fault is the root cause of a failure. That means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures.

For example, in a software system, an incorrectly written instruction in a program may decrement an internal variable instead of incrementing it. Clearly, if this statement is executed, it will result in the incorrect value being written. If other program statements then use this value, the whole system will deviate from its desired behavior. In this case, the erroneous statement is the fault, the invalid value is the error, and that if the variable is never read after being written, no failure will occur. Or, if the invalid statement is never executed, the fault will not lead to an error. Thus, the mere presence of errors or faults does not necessarily imply system failure.

Fault Classifications

Based on duration, faults can be classified as transient or permanent. A transient fault will eventually disappear without any apparent intervention, whereas a permanent one will remain unless it is removed by some external agency. While it may seem that permanent faults are more severe, from an engineering perspective, they are much easier to diagnose and handle.

Finally, based on how a failed component behaves once it has failed, faults can be classified into the following categories: **Crash faults :** the component either completely stops operating or never returns to a valid state.

Omission faults : the component completely fails to perform its service

Timing faults: the component does not complete its service on time

Byzantine faults: these are faults of an arbitrary nature.

Failure Models In Distributed Systems

Crash: Server halts, but was working ok until then, e.g. O.S. failure

(Distributed System)

Omission: Server fails to receive or respond or reply, e.g. server not listening or buffer overflow.

Timing: Server response time is outside its specification, client may give up. Response: Incorrect response or incorrect processing due to control flow out of synchronization.

Arbitrary value (or Byzantine): Server behaving erratically, for example providing arbitrary responses at arbitrary times. Server output is inappropriate but it is not easy to determine this to be incorrect. Duplicated message due to buffering problem may be given as an example. Alternatively, there may be a malicious element involved.

Agreement In Faulty Distributed Systems

Agreement in faulty distributed systems is used to elect a coordinator process or deciding to commit a transaction in distributed systems. It uses a majority voting mechanism which can tolerate K faulty out of $2K+1$ processes. (K fails, K+1 majority OK)

Byzantine Generals Problem

In this example, Enemy Red Army, as before, but Blue Army is under control of N generals (encamped separately). M (unknown) out of N generals are traitors and will try to prevent the N-M loyal generals reaching agreement. Communication is reliable by one to one telephone between pairs of generals to exchange troop strength information.

Problem: How can the blue army loyal generals reach agreement on troop strength of all other loyal generals?

Postcondition: If the i th general is loyal then $\text{troops}[i]$ is troop strength of general i . If the i th general is not loyal then $\text{troops}[i]$ is undefined (and is probably incorrect).

Algorithm

Each general sends a message to the N-1 (i.e. 3) other generals. Loyal generals tell truth, traitors lie. The results of message exchanges are collated by each general to give vector[N]. Each general sends vector[N] to all other N-1 (3) generals. Each general examines each element received from the other N-1 look for the majority response for each blue general. Algorithm works since traitor generals are unable to affect messages from loyal generals. Overcoming M traitor generals requires a minimum $2M+1$ loyal (3M+1) generals in total).

Recovery : Once failure has occurred in many cases, it is important to recover critical processes to a known state in order to resume processing. Problem is compounded in distributed systems. There are two approaches for the recovery in distributed environments.

Backward recovery, by use of checkpointing (global snapshot of distributed system status) to record the system state but checkpointing is costly (performance degradation).

DS-93

Forward recovery attempt to bring system to a new stable state from which it is possible to proceed (applied in situations where the nature of errors are known and a reset can be applied).

Forward recovery is most extensively used in distributed systems and generally safest can be incorporated into middleware layers, complicated in the case of process, machine or network failure. It gives no guarantee that same fault may occur again (deterministic view - affects failure transparency properties), and can not be applied to irreversible(non-idempotent) operations, e.g. ATM withdrawal.

Prob. 11 Explain byzantine faults and agreement in detail. [R.T.U. 2019]

OR

Explain Byzantine agreement problem and suggest a solution to this problem. [R.T.U. 2016]

OR

Write short note on the Agreement protocols. [R.T.U. 2016]

OR

Write short note on Byzantine faults. [R.T.U. 2017, 2013]

OR

What do you understand by Byzantine agreement? Explain. [R.T.U. 2014]

OR

Write short note on Byzantine agreement. [R.T.U. 2015]

OR

Define Byzantine agreement problem with its solution. What do you mean by agreement protocol? [R.T.U. 2017, 2011]

OR

Explain system model for agreement protocols. Briefly explain Byzantine agreement problem with example. [R.T.U. 2010]

Sol. Byzantine Faults : Distributed systems are subject to a variety of faults and attacks. We consider general (Byzantine) faults, i.e. a faulty node may exhibit arbitrary behavior. In particular, a faulty node may corrupt its local state and send arbitrary messages, including specific messages aimed at subverting the system. Many security attacks, such as censorship, freeloading, misrouting, and data corruption, can be modeled as Byzantine faults.

Systems can be protected with Byzantine Fault Tolerance (BFT) techniques, which mask a bounded number of Byzantine faults, e.g. using state machine replication. BFT is a very powerful technique, but it has its costs. In a practical system that needs to tolerate up to f concurrent Byzantine faults, BFT cannot be implemented with less than $3f+1$ replicas.

Moreover, BFT scales poorly to large replica groups; as more servers are added, the throughput of the system may actually decrease.

We explore an alternative approach that aims at detecting rather than masking faulty behavior. With this approach, the system does not make any attempt to hide the symptoms of Byzantine faults. Rather, each node is equipped with a detector that monitors other nodes for signs of faulty behavior. If the detector determines that some node has become faulty, it notifies the application software, which can then take appropriate action. For example, nodes can cease to communicate with the faulty node; once all correct nodes have followed suit, the faulty node is isolated and the fault is contained.

Fault detection is weaker than masking. For instance, detection is insufficient for dealing with faults that have serious and irreversible effects, such as deletion of all copies of an important document. However, detection may offer an efficient and scalable alternative to BFT for faults that have limited or recoverable effects, including freeloading, censorship, and denial-of-service.

We are interested in fault detectors that provide accountability. With such detectors, each action is undeniably associated with the identity of the node that has performed the action, allowing the system to gather irrefutable evidence of faulty behavior.

The fault detection systems we consider should guarantee at least two properties. The system should be complete: whenever a correct node observes the effects of faulty behavior, the system eventually generates evidence against at least one faulty node. Also, the system should be accurate: it never generates valid evidence against a correct node.

Adding accountability to a distributed system has several important advantages, regardless of whether the systems uses BFT or not: first, any faulty behavior by a node is guaranteed to be detected. Second, the evidence produced by the detector can be used to convince third parties that a fault has occurred. Lastly, the presence of accountability alone deters certain types of attacks on a system, because it identifies and exposes faulty nodes.

Agreement Protocols : In distributed system where sites often compete as well as cooperate to achieve a common goal, it is required that sites reach **mutual agreement**. Reaching an agreement typically requires that sites have knowledge about the values of other sites when the system is free from failures, an agreement can easily be reached among the processor. However when the system is prone to failure,

this method does not work. This is because faulty processor can send conflicting values to other processors preventing them from reaching an agreement. In the presence of faults, processors must exchange their values with other processors and relay the value received from other processor several times to isolate the effect of faulty processor. This entire process of reaching an agreement is called an **agreement protocol**.

In agreement problems, non-faulty processors in a distributed system should be able to reach a common agreement, even if certain components in the system are faulty.

The agreement is achieved through an agreement protocol that involves several rounds of message exchange among the processors.

Agreement problems can be studied under the following system model:

- There are n processors in the system and at most m processor can be faulty.
- The processors can directly communicate with other processors by message passing.
- A receiver processor always knows the identity of the sender processor of the message.
- The communication medium is reliable, and only processors are prone to failures.

There are three well known agreement problems in distributed system : **the Byzantine agreement problem, the consensus problem and the interactive consistency problem.**

In **Byzantine agreement problem**, a single value, which is to be agreed on is initialized by an arbitrary processor and all non-faulty processors have to agree on that value.

In **consensus problem**, every processor has its own initial value and all non-faulty processors must agree on a single common value.

In **interactive consistency problem**, every process has its own initial value and all non-faulty processors must agree on a set of common values. The starting values and final agreement of three problems are given below:

Problem	Byzantine Agreement	Consensus	Interactive Consistency Problem
Who initiates the value	One processor	All Processors	All processors
Final agreement	Single value	Single value	A vector of values

Fig. : Three agreement problems

Byzantine Agreement Problem : The Byzantine Agreement protocol which aims at establishing a fault tolerant agreement when one or more nodes in a system have been compromised or failed.

The Byzantine general problem formulation is as follows :
Imagine that several divisions of Byzantine army are camped outside an enemy city. Each division commanded by its own general. The general can communicate, with one another only by messenger. After observing the enemy they must decide upon a common plan of action. However, some of the general may be traitors, trying to prevent the loyal general from reaching agreement. The generals must have an algorithm to guarantee that :

- All loyal generals decide upon the same plan of action: The loyal general will all do what the algorithm says they should, but the traitors may do anything they wish. The loyal generals should not only reach agreement, but should agree upon a reasonable plan we therefore also want to insure that.
- A small number of traitors cannot cause the loyal general to adopt a bad plan.

The Byzantine general problem was described as the design of an algorithm such that when the commanding general sends an order to his $(n - 1)$ lieutenant general then the algorithm guarantees that

- All loyal lieutenants obey the same order.
- If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

In Byzantine agreement problem, an arbitrarily chosen processor called the source processor broadcast its initial value to all other processor. The requirements are :

- Agreement :** All non-faulty processors agree on the same value.
- Termination :** Eventually each correct process sets its decision variable.
- Integrity :** If the source processor is non-faulty, then the common agreed upon value by all non-faulty processors should be initial value of the source. Note that for the Byzantine general problem, if the source processors is faulty, then all non-faulty processors can agree on any common value. It is irrelevant what value faulty processors agree on or whether they agree on a value at all.

Prob.12 Explain impossibility of consensus and randomized distributed agreement. [R.T.U. 2018]

OR

Write short notes on Randomized Distributed Agreement. [RTU 2015, 2014]

Sol. Consensus : A fundamental problem in distributed computing and multi-agent systems is to achieve overall

implementation
s means that
remote) of a
the case of a
passed from
f the object

erfaces, which

In CORBA's
ition language

defining their
et of named
ations. IDL is
lementations.
r to C++ and

mentation tells
ilable and how
is, the CORBA
ing languages.

Prob. 16 Explain communication in two-phase commit protocols. Define transaction recovery in distributed transaction. [RTU 2011]

Sol. During the process of a transaction, the only communication between coordinator and participant is the join request. The client request to commit or abort goes to the coordinator.

If client or participant request abort, the coordinator informs the participants immediately. It is when the client asks to commit the transaction that the 2PC comes into use. In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit and in the second it tells them to commit (or abort) the transaction. The coordinator in a distributed transaction communicates with the participant to carry out the two phase commit protocol by means of operations given below. The methods cancommit, docommit and doabort are methods in the interface of the participants. The methods havecommitted and getdecision are in coordinator phase.

(i) **canCommit(trans)** : Yes / No

This is a request with a reply

It is a call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

(ii) **doCommit(trans)**

It is a call from coordinator to participant to tell participant to commit its part of a transaction.

(iii) **doAbort(trans)** : These are asynchronous requests to avoid delays

It is a call from coordinator to participant to tell participant to abort its part of a transaction.

(iv) **haveCommitted(trans; participant)**-Asynchronous request

It is a call from participant to coordinator to confirm that it has committed the transaction.

(v) **getDecision(trans)** : Yes / No

It is a call from participant to coordinator to ask for the decision on a transaction after it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages.

- The two phase commit (2PC) protocol is similar to a real life example of voting scheme.
- Voting is initiated by the coordinator of a transaction. All participants in the distributed transaction must agree on a decision about whether to commit or abort the transaction.
- Before a participant can vote to commit, it must be ready to perform the commit. When all participants agree and are ready to commit, then transaction is committed.
- Each participants (including coordinator) maintains a private work space (stable storage) for keeping track of updated data objects.
- The updates that recorded in the stable storage are an activity log of a transaction. Each participants has a stable activity log which is necessary for the durability and reliability of a committed transaction.
- Fig illustrates the execution flow of a two phase commit atomic transaction. Precommit and commit are two synchronization points for each participants.

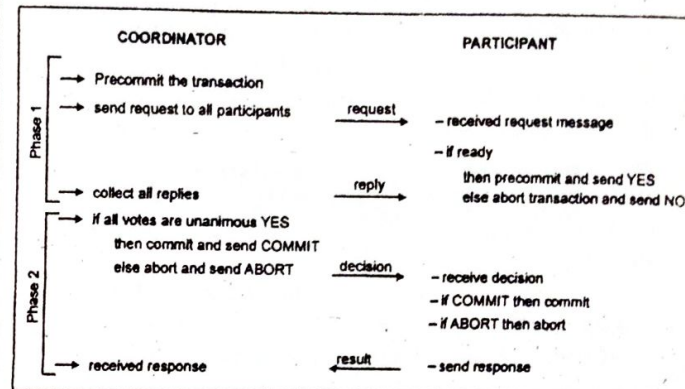


Fig. : Two-Phase Commit Atomic Transaction Protocol

Recovery from Distributed Transactions

If the database server faults while uncommitted operations are pending, it must either rollback or commit those operations on startup to preserve the atomic nature of the transaction.

If uncommitted operations from a distributed transaction are found during recovery, the database server attempts to connect to DTC and requests that it be re-enlisted in the pending or in-doubt transactions. Once the re-enlistment is complete, DTC instructs the database server to roll back or commit the outstanding operations.

If the re-enlistment process fails, SQL anywhere has no way of knowing whether the in-doubt operations should be committed or rolled back and recovery fails. If we want the database in such a state to recover, regardless of the uncertain state of the data, we can force recovery using the following database server options :

- **-tmf** : If DTC cannot be located, the outstanding operations are rolled back and recovery continues.
- **-tmt** : If re-enlistment is not achieved before the specified time, the outstanding operations are rolled back and recovery continues.

□□□