Name: Shivam Tiwari
Roll No: 5117060

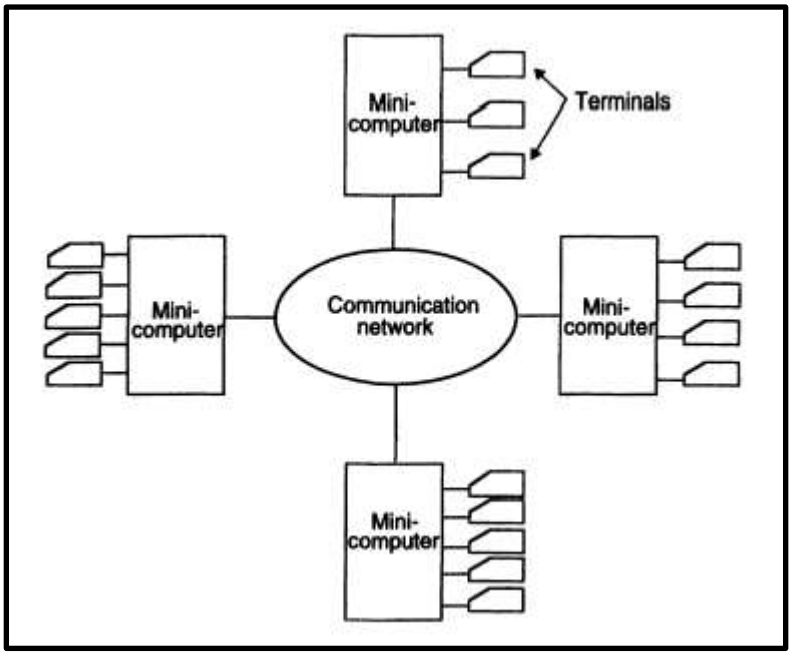
DISTRIBUTED COMPUTING ASSIGNMENT

Q1. Explain different types of distributed computing systems

Ans.

1. Mini computer Model

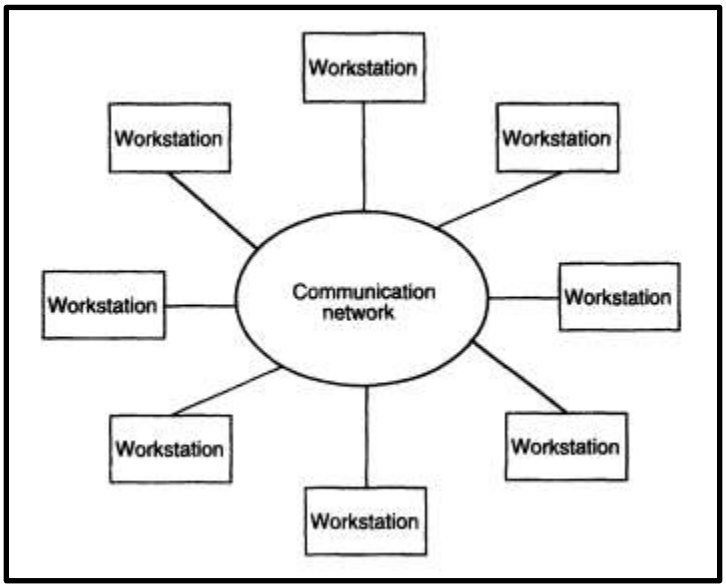
The minicomputer model is a simple extension of the centralized time-sharing system. A distributed computing system based on this model consists of a few minicomputers (they may be large supercomputers as well) interconnected by a communication network. Each minicomputer usually has multiple users simultaneously logged on to it. For this, several interactive terminals are connected to each minicomputer. Each user is logged on to one specific minicomputer, with remote access to other minicomputers. The network allows a user to access remote resources that are available on some machine other than the one on to which the user is currently logged.



2. Workstation Model

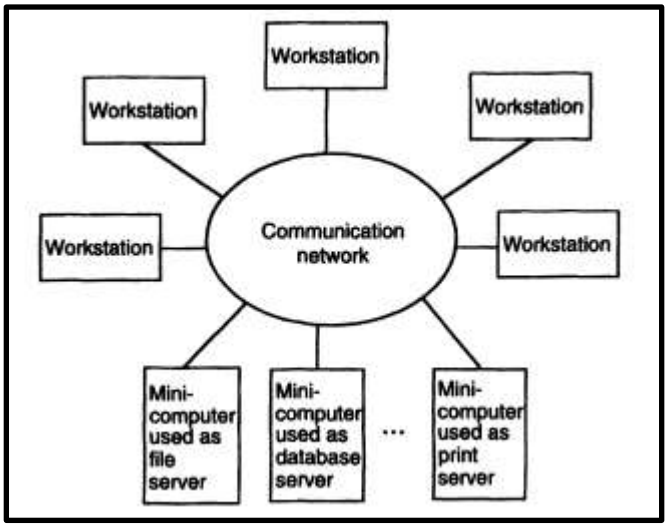
A distributed computing system based on the workstation model consists of several workstations interconnected by a communication network. The idea of the workstation model is to interconnect all these workstations by a high-speed LAN so that idle workstations may be used to process jobs of users who are logged onto other workstations and do not have sufficient processing power at their own workstations to get their jobs processed efficiently.

In this model, a user logs onto one of the workstations called his or her "home" workstation and submits jobs for execution. When the system finds that the user's workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the processes from the user's workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user's workstation.



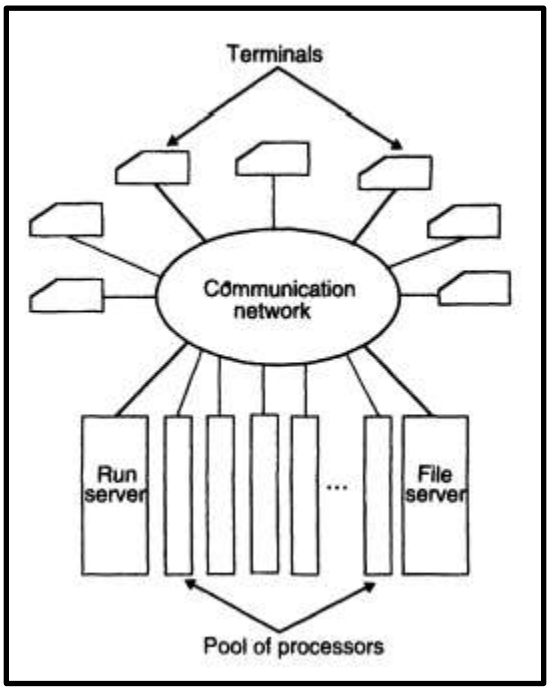
3. Workstation-server Model

The workstation model is a network of personal workstations, each with its own disk and a local file system. A workstation with its own local disk is usually called a diskful workstation and a workstation without a local disk is called a diskless workstation. With the proliferation of high-speed networks, diskless workstations have become more popular in network environments than diskful workstations, making the workstation-server model more popular than the workstation model for building distributed computing systems.



4. Processor-pool Model

The processor-pool model is based on the observation that most of the time a user does not need any computing power but once in a while he or she may need a very large amount of computing power for a short time (e.g., when recompiling a program consisting of a large number of files after changing a basic shared declaration). Therefore, unlike the workstation-server model in which a processor is allocated to each user, in the processor- pool model the processors are pooled together to be shared by the users as needed.



5. Hybrid Model

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

Q2. Explain different types of transparencies.

Ans.

a. Access Transparency

Access transparency means that users should not need or be able to recognize whether a resource (hardware or software) is remote or local. This implies that the distributed operating

system should allow users to access remote resources in the same way as local resources. That is, the user interface, which takes the form of a set of system calls, should not distinguish between local and remote resources, and it should be the responsibility of the distributed operating system to locate the resources and to arrange for servicing user requests in a user-transparent manner.

b. Location Transparency

The two main aspects of location transparency are as follows:

1. Name transparency

This refers to the fact that the name of a resource (hardware or software) should not reveal any hint as to the physical location of the resource. That is, the name of a resource should be independent of the physical connectivity or topology of the system or the current location of the resource. Furthermore, such resources, which are capable of being moved from one node to another in a distributed system (such as a file), must be allowed to move without having their names changed. Therefore, resource names must be unique systemwide.

2. User mobility

This refers to the fact that no matter which machine a user is logged onto, he or she should be able to access a resource with the same name. That is, the user should not be required to use different names to access the same resource from. two different nodes of the system. In a distributed system that supports user mobility, users can freely log on to any machine in the system and access any resource without making any extra effort. Both name transparency and user mobility requirements call for a systemwide, global resource naming facility

c. Replication Transparency

For better performance and reliability, almost all distributed operating systems have the provision to create replicas (additional copies) of files and other resources on different nodes of the distributed system. In these systems, both the existence of multiple copies of a replicated resource and the replication activity should be transparent to the users. That is, two 'important issues related to replication transparency are naming of replicas and replication control. It is the responsibility of the system to name the various copies of a resource and to map a user-supplied name of the resource to an appropriate replica of the resource. Furthermore, replication control decisions such as how many copies of the resource should be created, where should each copy be placed, and when should a copy be created/deleted should be made entirely automatically by the system in a user-transparent manner.

d. Failure Transparency

Failure transparency deals with masking from the users' partial failures in the system, such as a communication link failure, a machine failure, or a storage device crash. A distributed operating

system having failure transparency property will continue to function, perhaps in a degraded form, in the face of partial failures. For example, suppose the file service of a distributed operating system is to be made failure transparent. This can be done by implementing it as a group of file servers that closely cooperate with each other to manage the files of the system and that function in such a manner that the users can utilize the file service even if only one of the file servers is up and working. In this case, the users cannot notice the failure of one or more file servers, except for slower performance of file access operations. Any type of service can be implemented in this way for failure transparency. However, in this type of design, care should be taken to ensure that the cooperation among multiple servers does not add too much overhead to the system. Complete failure transparency is not achievable with the current state of the art in distributed operating systems because all types of failures cannot be handled in a user transparent manner. For example, failure of the communication network of a distributed system normally disrupts the work of its users and is noticeable by the users. Moreover, an attempt to design a completely failure-transparent distributed system will result in a very slow and highly expensive system due to the large amount of redundancy required for tolerating all types of failures. The design of such a distributed system, although theoretically possible, is not practically justified.

e. Migration Transparency

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

1. Migration decisions such as which object is to be moved from where to where should be made automatically by the system.

2. Migration of an object from one node to another should not require any change in its name.

3. When the migrating object is a process, the interprocess communication mechanism should ensure that a message sent to the migrating process reaches it without the need for the sender process to resend it if the receiver process moves to another node before the message is received.

f. Concurrency Transparency

In a distributed system, multiple users who are spatially separated use the system concurrently. In such a situation, it is economical to share the system resources (hardware or software) among the concurrently executing user processes. However, since the number of available resources in a computing system is restricted, one user process must necessarily influence the action of other concurrently executing user processes, as it competes for resources. For example, concurrent updates to the same file by two different processes should be prevented. Concurrency transparency means that each user has a feeling that he or she is the sole user of the system and other users do not exist in the system. For providing concurrency transparency, the resource sharing mechanisms of the distributed operating system must have the following four properties:

1. An event-ordering property ensures that all access requests to various system resources are properly ordered to provide a consistent view to all users of the system.
2. A mutual-exclusion property ensures that at any time at most one process accesses a shared resource, which must not be used simultaneously by multiple processes if program operation is to be correct.

3. A no-starvation property ensures that if every process that is granted a resource, which must not be used simultaneously by multiple processes, eventually releases it, every request for that resource is eventually granted.

4. A no-deadlock property ensures that a situation will never occur in which competing processes prevent their mutual progress even though no single one requests more resources than available in the system.

g. Performance Transparency

The aim of performance transparency is to allow the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system. As far as practicable, a situation in which one processor of the system is overloaded with jobs while another processor idle should not be allowed to occur. That is, the processing capability of the system should be uniformly distributed among the currently available jobs in the system. This requirement calls for the support of intelligent resource allocation and process migration facilities in distributed operating systems.

h. Scaling Transparency

The aim of scaling transparency is to allow the system to expand in scale without disrupting the activities of the users. This requirement calls for open-system architecture and the use of scalable algorithms for designing the distributed operating system components.

Q3. Explain implementation of RPC Mechanism.

Ans.

The RPC model is similar to the well-known and well-understood procedure call model used for the transfer of control and data within a program in the following manner:

1. For making a procedure call, the caller places arguments to the procedure in some well-specified location.

2. Control is then transferred to the sequence of instructions that constitutes the body of the procedure.

3. The procedure body is executed in a newly created execution environment that includes copies of the arguments given in the calling instruction.
4. After the procedure's execution is over, control returns to the calling point, possibly returning a result.

The RPC mechanism is an extension of the procedure call mechanism in the sense that it enables a call to be made to a procedure that does not reside in the address space of the calling process. The called procedure (commonly called remote procedure) may be on the same computer as the calling process or on a different computer. In the case of RPC, since the caller and the callee processes have disjoint address spaces (possibly on different computers), the remote procedure has no access to data and variables of the caller's environment. Therefore the RPC facility uses a message-passing scheme for information exchange between the caller and the callee processes. As shown in Figure 4.1, when a remote procedure call is made, the caller and the callee processes interact in the following manner:

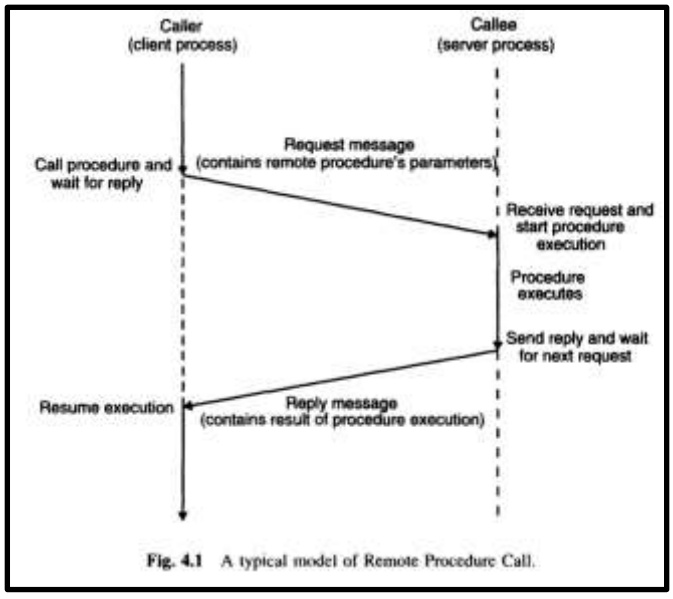


Fig. 4.1 A typical model of Remote Procedure Call.

1. The caller (commonly known as client process) sends a call (request) message to the callee (commonly known as server process) and waits (blocks) for a reply message. The request message contains the remote procedure's parameters, among other things.
2. The server process executes the procedure and then returns the result of procedure execution in a reply message to the client process.
3. Once the reply message is received, the result of procedure execution is extracted, and the caller's execution is resumed.

The server process is normally dormant, awaiting the arrival of a request message. When one arrives, the server processes the procedure's parameters, computes the result, sends a reply message, and then awaits the next call message. Note that in this model of RPC, only one of the two processes is active at any given time. However, in general, the RPC protocol makes no restrictions on the concurrency model implemented, and other models of RPC are possible depending on the details of the parallelism of the caller's and callee's environments and the RPC implementation. For example, an implementation may choose to have RPC calls to be

asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a thread to process an incoming request, so that the server can be free to receive other requests.

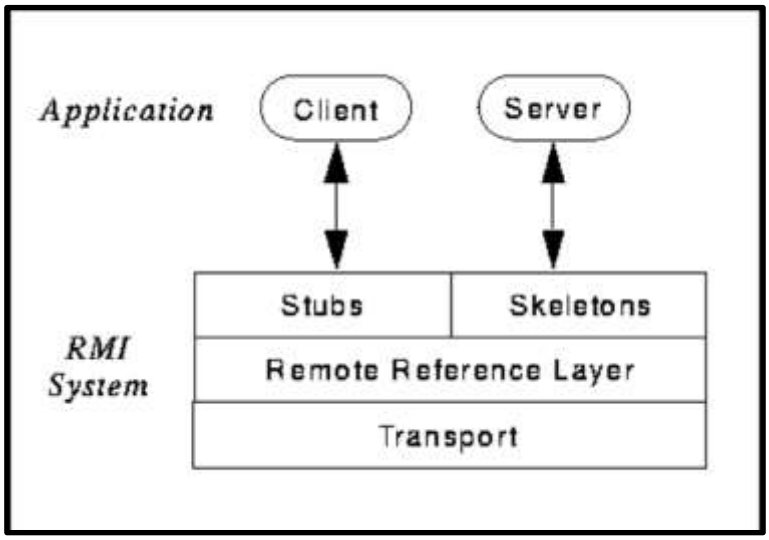
Q4. Describe the RMI architecture and its working in detail.

Ans.

A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

The RMI system consists of three layers:

- The stub/skeleton layer - client-side stubs (proxies) and server-side skeletons
- The remote reference layer - remote reference behavior (such as invocation to a single object or to a replicated object)
- The transport layer - connection set up and management and remote object tracking



1. The Stub/Skeleton Layer

The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. This layer does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of *marshal streams*. Marshal streams employ a mechanism called *object serialization* which enables Java objects to be transmitted between address spaces. Objects transmitted using the object serialization system are passed by copy to the remote address space, unless they are remote objects, in which case they are passed by reference.

A *stub* for a remote object is the client-side proxy for the remote object. Such a stub implements all the interfaces that are supported by the remote object implementation. A client-side stub is responsible for:

- Initiating a call to the remote object (by calling the remote reference layer).
- Marshaling arguments to a marshal stream (obtained from the remote reference layer).
- Informing the remote reference layer that the call should be invoked.
- Unmarshaling the return value or exception from a marshal stream.
- Informing the remote reference layer that the call is complete.

A *skeleton* for a remote object is a server-side entity that contains a method which dispatches calls to the actual remote object implementation. The skeleton is responsible for:

- Unmarshalling arguments from the marshal stream.
- Making the up-call to the actual remote object implementation.
- Marshaling the return value of the call or an exception (if one occurred) onto the marshal stream.

The appropriate stub and skeleton classes are determined at run time and are dynamically loaded as needed, as described in Dynamic Class Loading. Stubs and skeletons are generated using the rmic compiler.

2. The Remote Reference Layer

The remote reference layer deals with the lower-level transport interface. This layer is also responsible for carrying out a specific remote reference protocol which is independent of the client stubs and server skeletons.

Each remote object implementation chooses its own remote reference subclass that operates on its behalf. Various invocation protocols can be carried out at this layer. Examples are:

- Unicast point-to-point invocation.
- Invocation to replicated object groups.
- Support for a specific replication strategy.
- Support for a persistent reference to the remote object (enabling activation of the remote object).
- Reconnection strategies (if a remote object becomes inaccessible).

The remote reference layer has two cooperating components: the client-side and the server-side components. The client-side component contains information specific to the remote server (or servers, if the remote reference is to a replicated object) and communicates via the transport to the server-side component. During each method invocation, the client and server-side components perform the specific remote reference semantics. For example, if a remote object is part of a replicated object, the client-side component can forward the invocation to each replica rather than just a single remote object.

In a corresponding manner, the server-side component implements the specific remote reference semantics prior to delivering a remote method invocation to the skeleton. This component, for example, would handle ensuring atomic multicast delivery by communicating with other servers in a replica group (note that multicast delivery is not part of the JDK 1.1 release of RMI).

The remote reference layer transmits data to the transport layer via the abstraction of a stream-oriented *connection*. The transport takes care of the implementation details of connections. Although connections present a streams-based interface, a connectionless transport can be implemented beneath the abstraction.

3.The Transport Layer

In general, the transport layer of the RMI system is responsible for:

- Setting up connections to remote address spaces.
- Managing connections.
- Monitoring connection "liveness."
- Listening for incoming calls.
- Maintaining a table of remote objects that reside in the address space.
- Setting up a connection for an incoming call.
- Locating the dispatcher for the target of the remote call and passing the connection to this dispatcher.

The concrete representation of a remote object reference consists of an endpoint and an object identifier. This representation is called a *live reference*. Given a live reference for a remote object, a transport can use the endpoint to set up a connection to the address space in which the remote object resides. On the server side, the transport uses the object identifier to look up the target of the remote call.

The transport for the RMI system consists of four basic abstractions:

- An *endpoint* is the abstraction used to denote an address space or Java virtual machine. In the implementation, an endpoint can be mapped to its transport. That is, given an endpoint, a specific transport instance can be obtained.
- A *channel* is the abstraction for a conduit between two address spaces. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel.
- A *connection* is the abstraction for transferring data (performing input/output).
- The *transport* abstraction manages channels. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces (the local address space and a remote address space). Given an endpoint to a remote address space, a transport sets up a channel to that address space. The transport abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system.

A transport defines what the concrete representation of an endpoint is, so multiple transport implementations may exist. The design and implementation also supports multiple transports per address space, so both TCP and UDP can be supported in the same virtual machine. Note that the RMI transport interfaces are only available to the virtual machine implementation and are not available directly to the application.

Working

A client invoking a method on a remote server object actually makes use of a *stub* or proxy for the remote object as a conduit to the remote object. A client- held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote object and forwards invocation requests to that server object via the remote reference layer. Stubs are generated using the rmic compiler.

The *remote reference layer* is responsible for carrying out the semantics of the invocation. For example, the remote reference layer is responsible for determining whether the server is a single object or is a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics-whether the server is a single object or is a replicated object requiring communications with its replicas.

Also handled by the remote reference layer are the reference semantics for the server. The remote reference layer, for example, abstracts the different ways of referring to objects that are implemented in

(a) servers that are always running on some machine, and
(b) servers that are run only when some method invocation is made on them (activation).
At the layers above the remote reference layer, these differences are not seen.

The *transport layer* is responsible for connection setup, connection management, and keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport's address space.

In order to dispatch to a remote object, the transport forwards the remote call up to the remote reference layer. The remote reference layer handles any server-side behavior that needs to occur before handing off the request to the server-side skeleton. The skeleton for a remote object makes an up call to the remote object implementation which carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer, and transport on the server side, and then up through the transport, remote reference layer, and stub on the client side.

Q5. Explain different communication models used in the distributed systems.

Ans.

The 4 models of communication used in distributed systems are: RPC, RMI, MOM and Stream Oriented

1. RPC (Remote Procedure Call)

A remote procedure call is an interprocess communication technique that is used for client-server based applications. It is also known as a subroutine call or a function call. A

client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

2. RMI (Remote Method Invocation)

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM. RMI is used to build distributed applications; it provides remote communication between Java programs.

3. MOM (Message Oriented Middleware)

Message-oriented middleware (MOM) is software or hardware infrastructure supporting sending and receiving messages between distributed systems. MOM allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols. The middleware creates a distributed communications layer that insulates the application developer from the details of the various operating systems and network interfaces. APIs that extend across diverse platforms and networks are typically provided by MOM.

4. Stream Oriented

Stream-oriented communication is a form of communication in which timing plays a crucial role. Eg. audio stream. ... For example, text is encoded as ASCII or Unicode, images as GIF or lPEG, Audio streams as 16-bit samples using PCM.

Q6. Explain the message oriented, transient & persistent communication.

Ans.

A. Message Oriented Transient Communication:

Berkeley Socket Primitives, and Message-Passing Interface (MPI) are examples of message-oriented transient communication. These systems do not use on-disk queues for message storing. Below, we map the MPI primitive functions to one of the above types. MPI bsend simply appends the outgoing message to the local send buffer, and then keeps running. Thus, it is transient asynchronous communication. MPI sends a message and waits until copied to the remote buffer. This is similar to delivery-based transient synchronous communication. MPI send waits until receipt starts and MPI sendrecv waits for a reply, classifying each to receipt based and reply

based transient synchronous communication respectively. Hence, these constructs map very well to the 6 different communication types mentioned above. MPI allows you to choose what kinds of combination of Sync communication you wish to use. It is a much richer communication system where the programmer can choose what form of communication to use from a spectrum of choices.

Message Oriented Persistent Communication

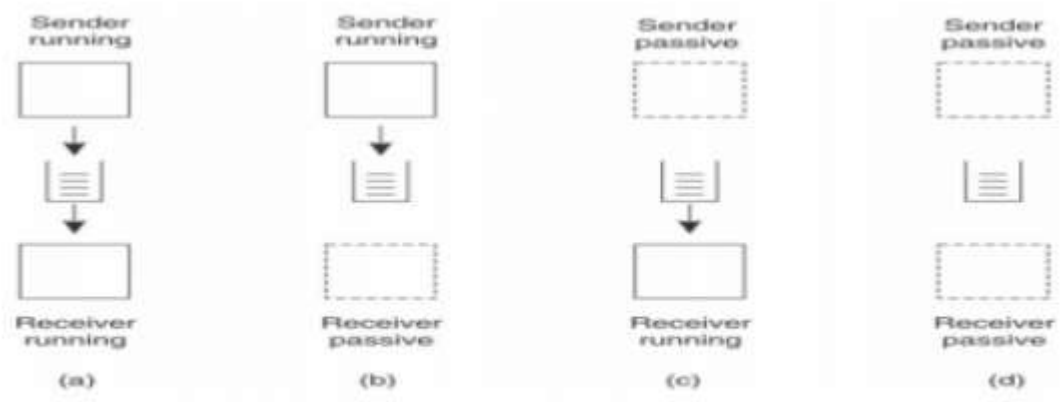


Figure 10.2: Various use cases of a Message Queuing Model

This class of communication is called Message Queuing Systems (MQS). They use on-disk buffers to ensure persistence of messages over long periods of time. Common open-source examples include Kafka, RabbitMQ, ActiveMQ, ZeroMQ, etc. An application level example for this is emails. All emails are stored on disks til delivery. The queues have names/addresses which are used to explicitly put messages into the queue. This is a form of store-and-forward communication. Persistence neither guarantees the duration in which the message will be delivered, nor the reading of the message. It only ensures delivery. Thus, it is a loosely coupled communication. Figure 10.2 explains the working of an MQS. The figure represents a series of queues(doing hop-by hop delivery) as a single abstract persistent queue.

Dark boxes indicate execution, while dotted boxes indicate that the system is not running. The first case (a) is the simplest where both A and B are running, and the message is delivered immediately. In case (b) where B is not running, the message is stored at the queue and B takes delivery at a later point. Example - email. In case (c), the sender is not running when B receives the message. Example - Reading email In case (d), the message is queued for an arbitrary amount of time as both A and B are passive. This is loose communication because the sender and the receiver are not participating in the communication at the same time. The typical abstractions used in the message queuing model are get and put. put allows us to insert into a specific queue. get removes a message from the head of the queue. get and put are analogous to send and receive. There is a non-blocking poll as well, which checks the queue for a message and returns the message if the queue is non-empty. It is a periodic process (might lead to resource wastage). Example - email clients checking servers. notify (also non-blocking) is an async message from the queue to the receiver notifying it of a message arrival.

Q7. What is the need for synchronization ? Present few application scenarios where physical clock synchronization is needed for a distributed system.

Ans.

Need for Synchronization:

A distributed system consists of a collection of distinct processes that are spatially separated and run concurrently. In systems with multiple concurrent processes, it is economical to share the system resources (hardware or software) among the concurrently executing processes. In such a situation, sharing may be cooperative or competitive. That is, since the number of available resources in a computing system is restricted, one process must necessarily influence the action of other concurrently executing processes as it competes for resources. For example, for a resource (such as a tape drive) that cannot be used simultaneously by multiple processes, a process willing to use it must wait if another process is using it. At times, concurrent processes must cooperate either to achieve the desired performance of the computing system or due to the nature of the computation being performed. Typical examples of process cooperation involve two processes that bear a producer-consumer or client-server relationship to each other. For instance, a client process and a file server process must cooperate when performing file access operations . Both cooperative and competitive sharing require adherence to certain rules of behavior that guarantee that correct interaction occurs. The rules for enforcing correct interaction are implemented in the form of synchronization mechanisms.

Application Scenario where physical clock synchronization is needed for a distributed system:

Every computer needs a timer mechanism (called a computer clock) to keep track of current time and also for various accounting purposes such as calculating the time spent by a process in CPU utilization, disk I/O, and so on, so that the corresponding user can be charged properly. In a distributed system, an application may have processes that concurrently run on multiple nodes of the system. For correct results, several such distributed applications require that the clocks of the nodes are synchronized with each other. For example, for a distributed on-line reservation system to be fair, the only remaining seat booked almost simultaneously from two different nodes should be offered to the client who booked first, even if the time difference between the two bookings is very small. It may not be possible to guarantee this if the clocks of the nodes of the system are not synchronized.

Q8. What are the disadvantages of token based mutual algorithms?

Ans.

The disadvantages of token based mutual exclusion algorithms are:

1. **Process failure**

A process failure in the system causes the logical ring to break. In such a situation, a new logical ring must be established to ensure the continued circulation of the token among other processes.

This requires detection of a failed process and dynamic reconfiguration of the logical ring when a failed process is detected or when a failed process recovers after failure.

2. **Lost token**

If the token is lost, a new token must be generated. Therefore, the algorithm must also have mechanisms to detect and regenerate a lost token.

3. **Starvation of remote processes**

As we know that in token based mutual exclusion algorithms a process having the token can only execute in the critical section. Most of the time the nodes or the processes follow a greedy approach and execute itself in the critical section irrespective of the requests that are present in their request buffer. This may lead to starvation of remote processes that have requested for the token.

4. **Detection and Regeneration of Lost Token is Difficult**

When passing tokens from one process to another, if there is a loss of reference of tokens, it becomes difficult to detect the lost token. If lost regeneration of the token and informing all the processes about the new token is not feasible.