CHAPTER 2

Module 2

Communication

Syllabus

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

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

2.1 Layered Protocols

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

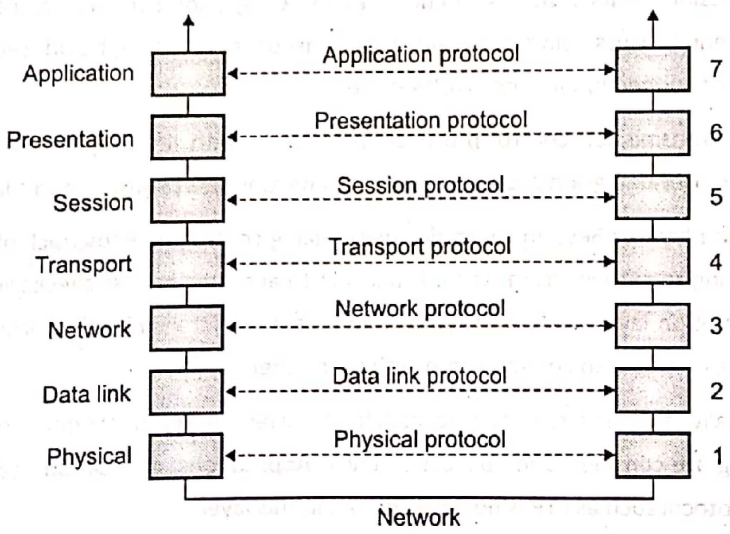


Fig. 2.1.1: Interfaces, Layers and Protocols in OSI model

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

2.1.1 Lower Layer Protocols

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

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

2.2 Interprocess Communication (IPC)

2.2.1 Types of Communication

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

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

2.2.2 Message Passing Interface

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

2.3 Remote Procedure Call (RPC)

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

2.3.1 RPC Operation

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

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

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

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

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

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

Step 5 : Server stub calls the server procedure.

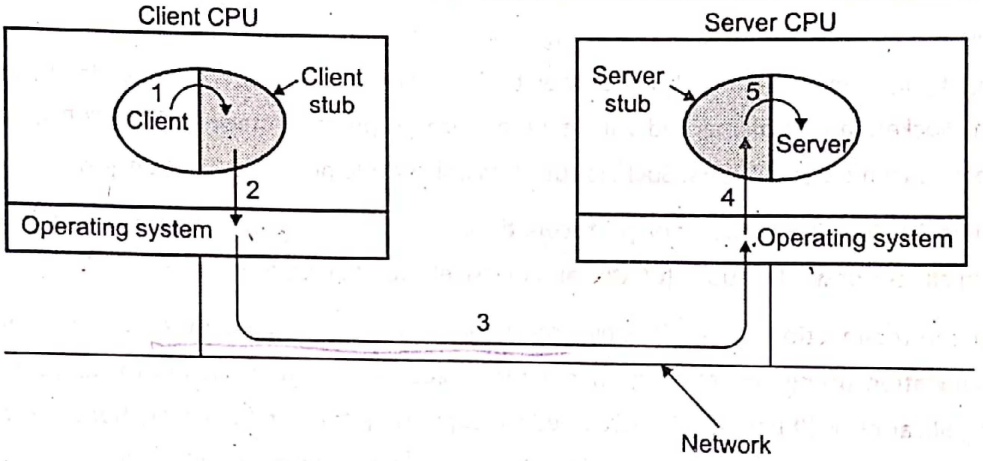


Fig. 2.3.1 : Steps in RPC

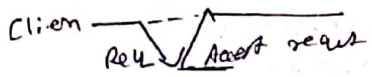
- The server then handover the result to server stub which packs it into message. Kernel then send the message to client where kernel at client handover it to client stub. Client stub unpack the message and handover result to client.

2.3.2 Implementation Issues

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

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

2.3.3 Asynchronous RPC



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

2.3.4 DCE RPC

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

Services provided by DCE are :

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

Goals and Working of DCE RPC

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

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

Steps in Writing a Client Server Application in DCE RPC

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

Binding Client to Server in DCE RPC

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

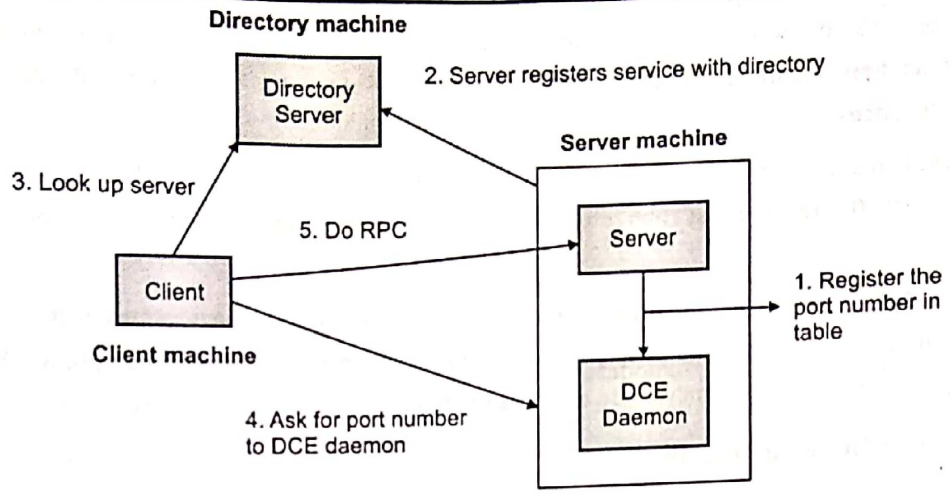


Fig. 2.3.2

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

2.4 Remote Object Invocation

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

2.4.1 Distributed Objects (RMI : Remote Method Invocation)

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

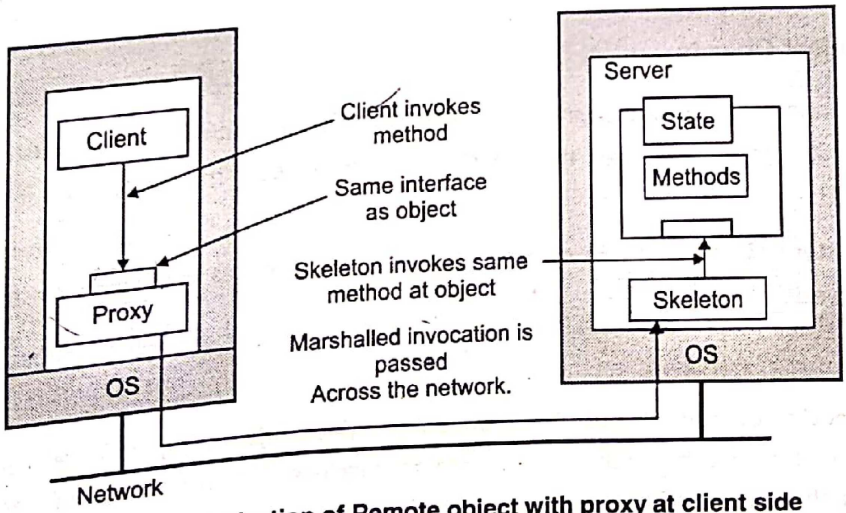


Fig. 2.4.1: Organization of Remote object with proxy at client side

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

Compile-time Versus Runtime objects

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

Persistent and Transient Objects

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

Object References

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

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

Parameter Passing

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

Example : Java RMI

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

The java Distributed Object Model

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

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

Java Remote Object Invocation

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

2.5 Message-Oriented Communication

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

2.5.1 Persistence Synchronicity in Communication

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

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

2.5.2 Combination of Communication Types

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

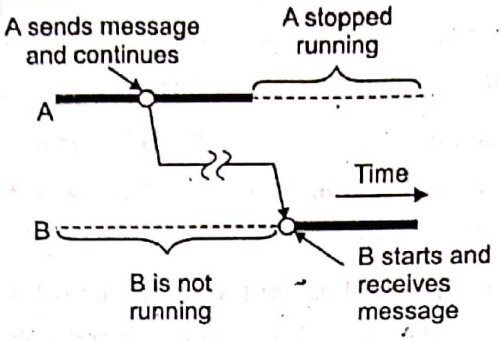


Fig. 2.5.1

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

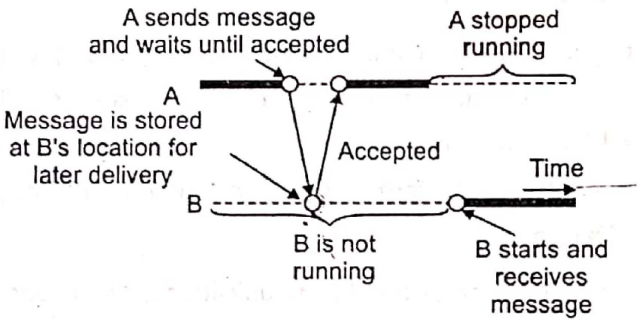


Fig. 2.5.2

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

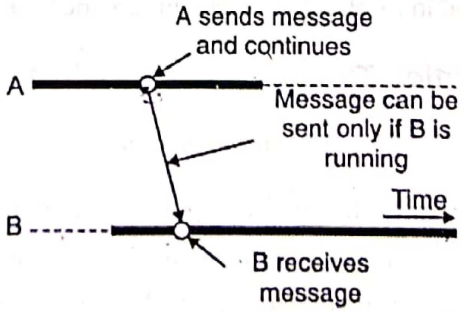


Fig. 2.5.3

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

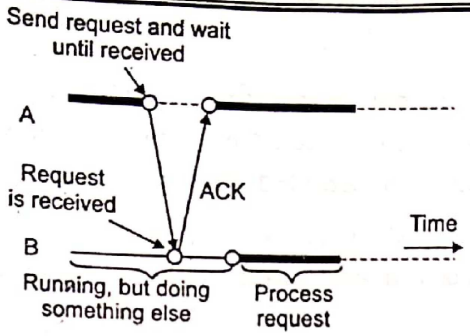


Fig. 2.5.4

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

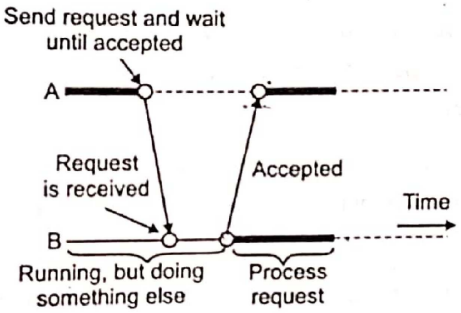


Fig. 2.5.5

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

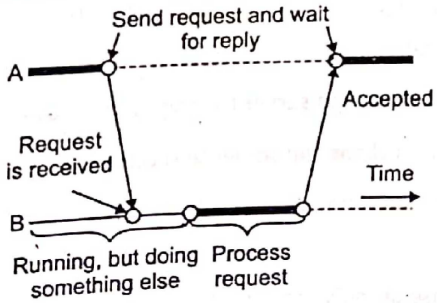


Fig. 2.5.6

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

2.5.3 Message-Oriented Transient Communication

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

Berkley Sockets

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

Message Passing Interface

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

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

2.5.4 Message-Oriented Persistent Communication

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

Message-Queuing Model

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

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

General Architecture of a Message-Queuing System

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

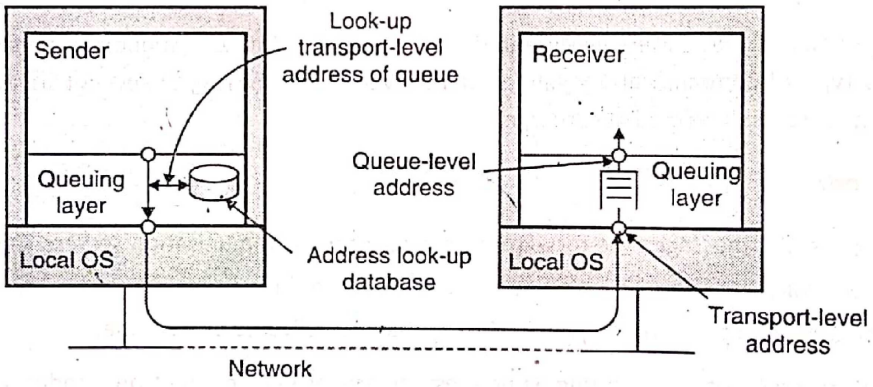


Fig. 2.5.7

Message Brokers

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

Example : IBM's WebSphere Message-Queuing System

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

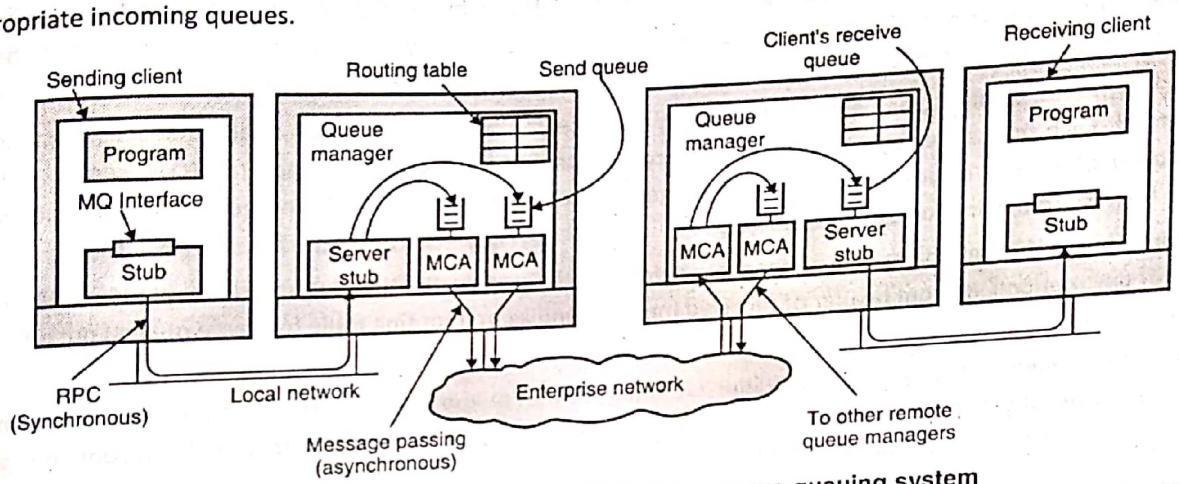


Fig. 2.5.8 : General organization of IBM's message-queuing system

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

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

Message Transfer

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

2.6 Stream-Oriented Communication

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

2.6.1 Continuous Media Support

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

Data Stream

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

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

2.6.2 Streams and Quality of Service (QoS)

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

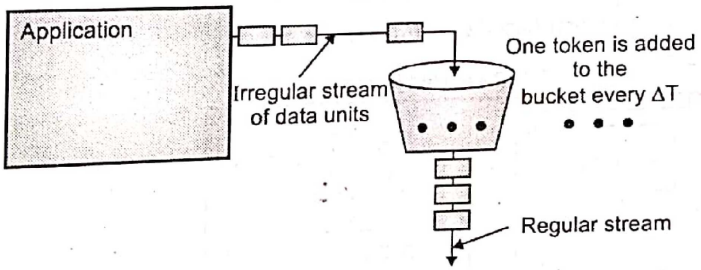


Fig. 2.6.1: Token Bucket Algorithm

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

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

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

Setting up a Stream

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

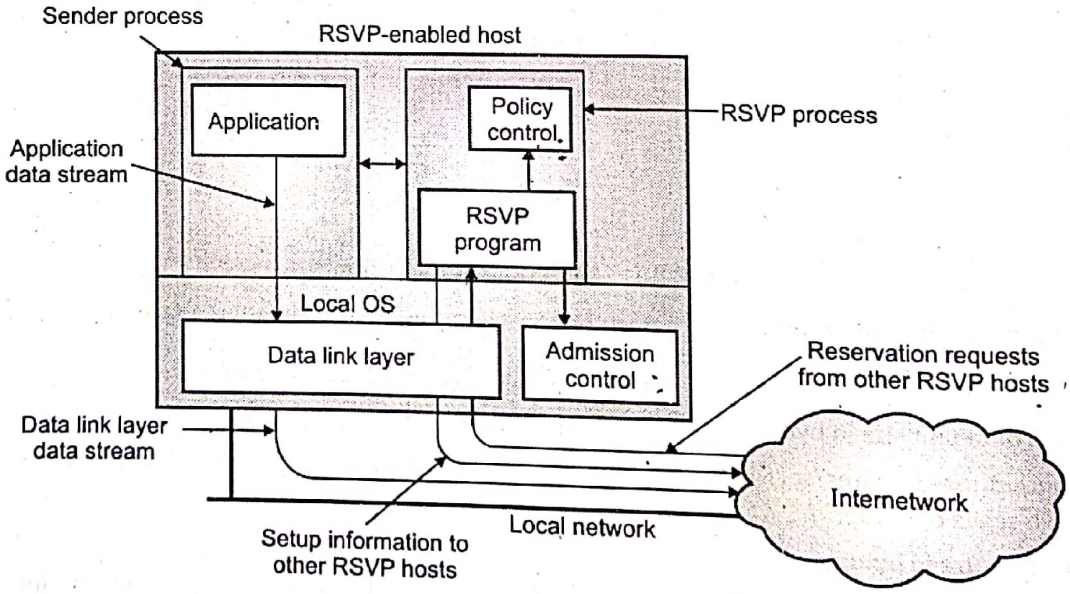


Fig. 2.6.2 : RSVP Protocol

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

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

2.6.3 Stream Synchronization

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

Synchronization Mechanisms

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

2.7 Group Communication

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

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

2.7.1 Application-Level Multicasting

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

Scribe : an Application Level Multicasting Scheme

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

Overlay Construction

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

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

2.7.2 Gossip-Based Data Dissemination

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

Information Dissemination Models

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

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

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

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

Removing Data

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

Applications of Epidemic Protocols

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

Review Questions

Q. 1 Explain ISO OSI model of communication?

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

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

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

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

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

Q. 7 What are the goals of DCE RPC?

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

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

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

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

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

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

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

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

Q. 16 What is object reference? Explain.

Q. 17 Write short note on "Java RMI"

Q. 18 Explain parameter passing in RMI.

Q. 19 Explain JAVA distributed object model.

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

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

Q. 22 Explain Berkley Sockets.

Q. 23 Explain MPI primitives.

Q. 24 Explain Message-Queuing Model.

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

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

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

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

Q. 29 What is discrete and continuous media.

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

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

Q. 32 Explain stream synchronization in detail.

Q. 33 Explain Application-Level Multicasting.

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

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

□□□