Chapter Four
Communication in Distributed Systems

4.1. Introduction

The communication in a distributed system (i.e., a multicomputer environment) is similar to communication in the situation between processes through a shared memory or message passing (IPC) on a uniprocessor (or multiprocessor) environment. The main difference is that in a distributed system, processes running on separate computers cannot directly access each other's memory. Nevertheless, processes in a distributed system can still communicate through either shared memory or message passing.

This chapter is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system, examining, in particular, the remote invocation paradigms.

Request-reply protocols represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing. In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation, and also provide direct support for RPC, RMI, MOM discussed below.

- ✓ The earliest and perhaps the best-known example of a more programmer-friendly model was the extension of the conventional procedure call model to distributed systems (the *remote procedure call*, or RPC, model), which allows client programs to call procedures transparently in server programs running in separate processes and generally in different computers from the client.
- ✓ Object-based programming model was extended to allow objects in different processes to communicate with one another by means of *remote method invocation* (RMI). RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

As you have learnt on the previous chapter i.e. chapter 2, communication between entities in the client-server or p2p system is achieved through a connection-oriented or connectionless protocol.

4.2. Types of Communication

There are a number of alternative ways, or modes, in which communication can take place. It is important to know and understand these different modes, because they are used to describe the different services that a communication subsystem offers to higher layers.

4.2.1. Data-oriented communication

In this mode, communication serves solely to exchange data between processes. Although the data might trigger an action at the receiver, there is no explicit transfer of control implied in this mode.

4.2.2. Control-oriented communication

Control-oriented communication explicitly associates a transfer of control with every data transfer. Data-oriented communication is clearly the type of communication used in communication via shared address space and shared memory, as well as message passing. Control-oriented communication is the mode used by abstractions such as remote procedure call, remote method invocation, active messages, etc.

4.2.3. Synchronous communication

In synchronous communication, the sender of a message blocks until the message has been received by the intended recipient. Synchronous communication is usually even stronger than this in that the sender often blocks until the receiver has processed the message and the sender has received a reply.

4.2.4. Asynchronous communication

In asynchronous communication, the sender continues execution immediately after sending a message (possibly without having received an answer). Another possible alternative involves the buffering of communication. In the buffered case, a message will be stored if the receiver is not able to pick it up right away. In the unbuffered case the message will be lost.

4.2.5. Transient communication

In transient communication, a message will only be delivered if a receiver is active. If there is no active receiver process (i.e., no one interested in or able to receive messages) then an undeliverable message will simply be dropped.

4.2.6. Persistent communication

In persistent communication, a message will be stored in the system until it can be delivered to the intended recipient.

As Figure 4.1 shows, all combinations of synchronous/asynchronous and transient/persistent are possible.

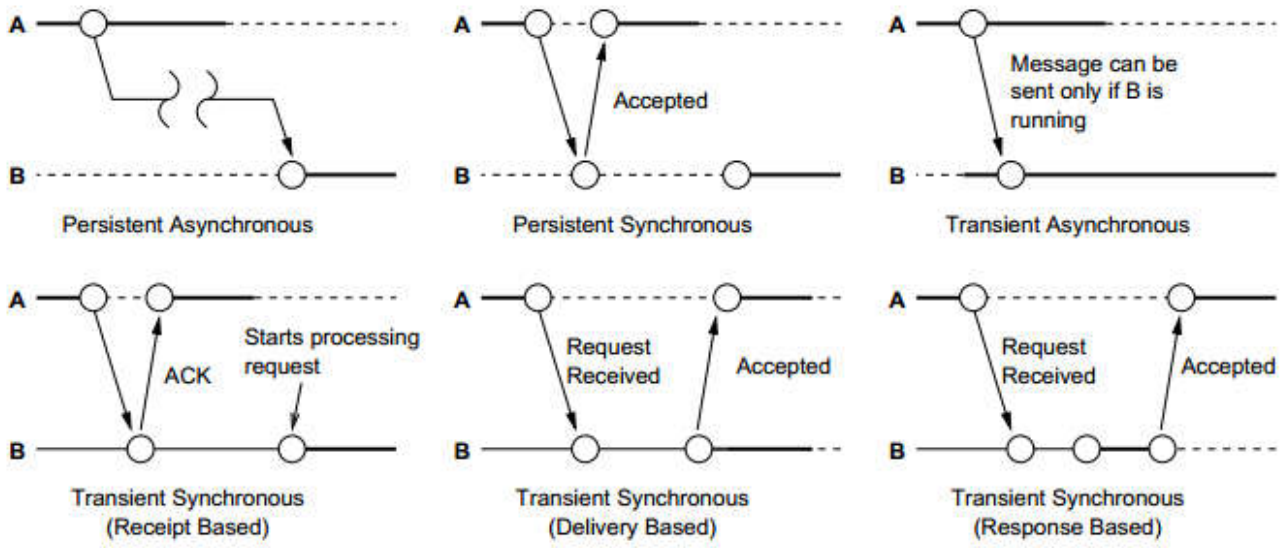

Figure 4.1 Possible combinations of synchronous/asynchronous and transient/persistent communication

4.3. Middleware protocols

Middleware is an application that logically lives (mostly) in the OSI application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications. The session and presentation layers in the OSI model have been replaced by a single middleware layer that contains application-independent protocols. These protocols do not belong to the lower layers. As it is shown in Figure 4.2, the network and transport service have been grouped into communication service as normally offered by an operating system, which, in return, manages the specific lowest-level hardware used to establish communication.

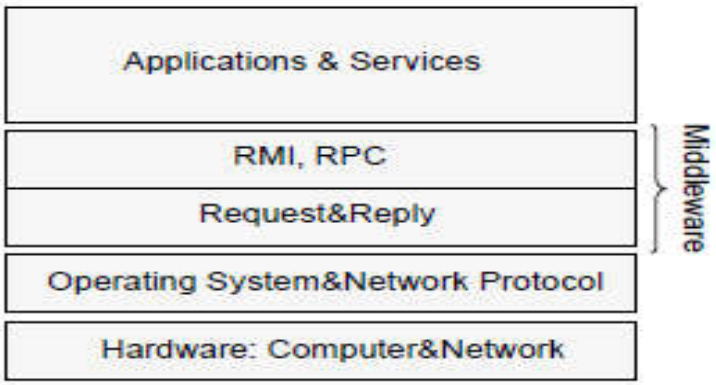

Figure 4.2: Communication Models and their Layered Implementation:

Middleware (Request-Replay, RPC, and RMI) and distributed applications have to be implemented on top of a network protocol. Such a protocol is implemented as several layers.

The following are the techniques/protocols that are used to establish communication between processes and objects of client and server remotely in distributed systems.

4.4. Request-Reply Protocol Communication in a Client-Server Model

The system is structured as a group of processes (objects), called *servers* that deliver services to *clients*.

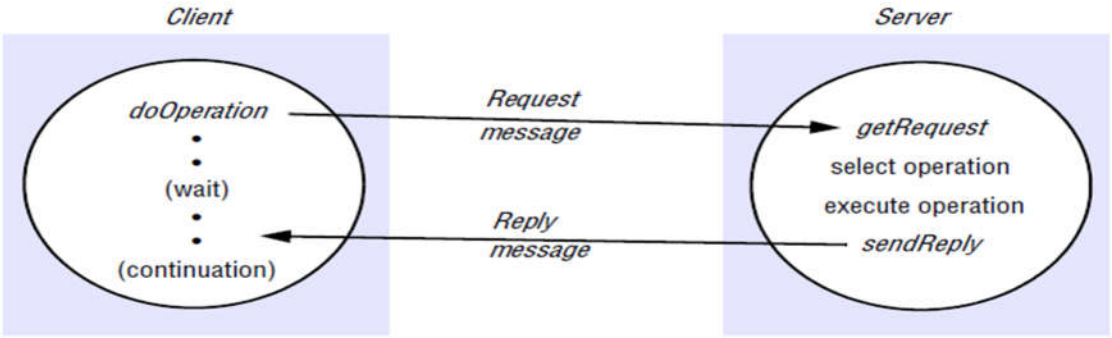

Figure 4.3: Request-Reply Protocol

Operations of the request-reply protocol are:

- *public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)*
  - ✓ Sends a request message to the remote server and returns the reply. The arguments specify the remote server, the operation to be invoked and the arguments of that operation.
- *public byte[] getRequest ();*
  - ✓ Acquires a client request via the server port.
- *public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);*
  - ✓ Sends the reply message to the client at its Internet address and port.

The client: send (request) to server_reference; receive(reply)

The server: receive(request) from client-reference; *execute requested operation, s*end (reply) to client_reference;

Request and *reply* are implemented based on the network protocol, for example in case of the Internet TCP (Transport Control Protocol) or UDP (User Datagram Protocol). Both TCP and UDP are transport protocols which implemented on top of the Internet protocol (IP).

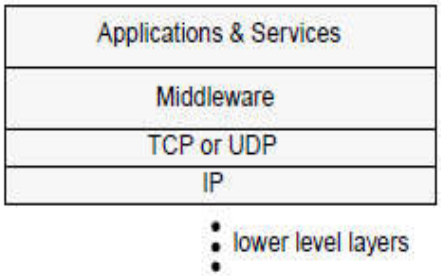

Figure 4.4: TCP and UDP layers

TCP is a reliable protocol - TCP implements additional mechanisms on top of IP to meet reliability guarantees.

UDP is a protocol that does not guarantee reliable transmission - UDP offers no guarantee of delivery.

HTTP: An example of a request-reply protocol: Hyper Text Transfer Protocol (HTTP) used by web browser clients to make requests to web servers and to receive replies from them.

HTTP is implemented over TCP. In the original version of the protocol, each client- server interaction consisted of the following steps:

- The client requests and the server accept a connection at the default server port or at a port specified in the URL.
- The client sends a request message to the server.
- The server sends a reply message to the client.
- The connection is closed.

4.5. Remote Procedure Call (RPC)

A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution), unless the client sends an asynchronous request to the server.

An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems. Also, callers generally must deal with such failures without knowing whether the remote procedure was actually invoked. Idempotent procedures (those that have no additional effects if called more than once) are easily handled, but enough difficulties remain that code to call remote procedures is often confined to carefully written low-level subsystems.

The concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing, with the goal of making the programming of distributed systems look similar, if not identical, to conventional programming – ***that is, achieving a high level of distribution transparency.***

Transparency: The originators of RPC, aimed to make remote procedure calls as much like local procedure calls as possible, with no distinction in syntax between a local and a remote procedure call. All

the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call.

Although request messages are retransmitted after a timeout, this is transparent to the caller to make the semantics of remote procedure calls like that of local procedure calls.

More precisely, RPC strives to offer at least location and access transparency, hiding the physical location of the (potentially remote) procedure and also accessing local and remote procedures in the same way.

The goal is make, for the programmer, distributed computing look like centralized computing.

- ***RPC is transparent***: the calling object (procedure) is not aware that the called one is executing on a different machine, and vice versa.

Sequence of events during an RPC

Step 1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.

Step 2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.

Step 3. The client's local operating system sends the message from the client machine to the server machine.

Step 4. The local operating system on the server machine passes the incoming packets to the server stub.

Step 5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.

Step 6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

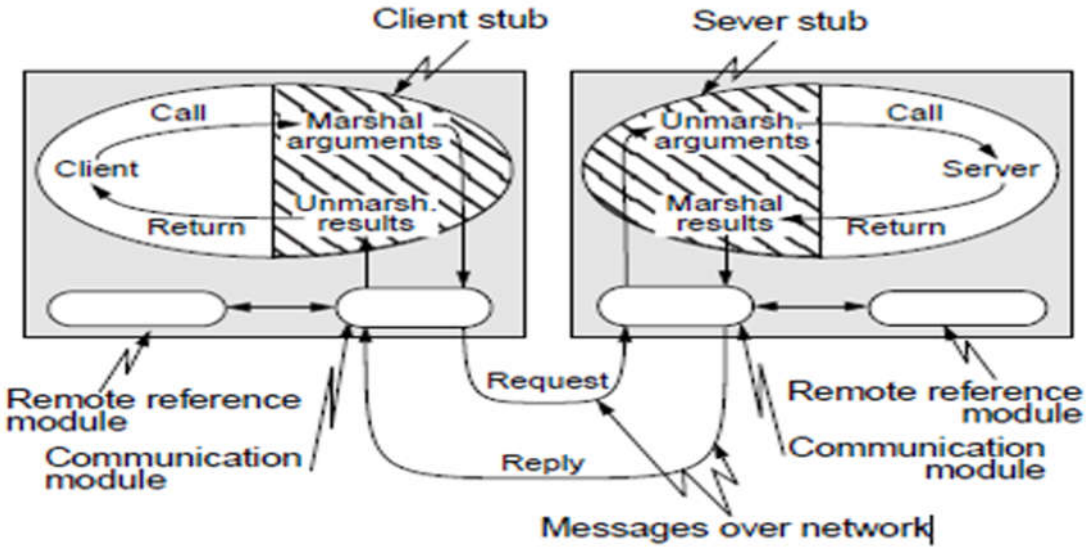

Figure 4.5: Remote Procedure Call Architectural Model

The client that accesses a service includes one *stub procedure* for each procedure in the service interface. The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its

communication module to the server. When the reply message arrives, it unmarshals the results. The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface. The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. The server stub procedure then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message. The service procedures implement the procedures in the service interface. The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.

4.6. Remote Method Invocation (RMI)

Remote method invocation (RMI) allows applications to call object methods loca ted remotely, sharing resources and processing load across systems. Unlike other systems for remote execution (RPC) which require that only simple data types or defined structures be passed to and from methods, RMI allows any Java object type to be used - even if the client or server has never encountered it before. RMI allows both client and server to dynamically load new object types as required.

Java RMI is a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism.

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

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

- The programmer is able to use the full expressive power of object-oriented programming in the development of distributed systems software, including the use of objects, classes and inheritance, and can also employ related object- oriented design methodologies and associated tools.
- Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.
- The issue of parameter passing is particularly important in distributed systems. RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference. Passing references is particularly attractive if the underlying parameter is large or complex. The remote end, on receiving an object reference, can then access this object using remote method invocation, instead of having to transmit the object value across the network.

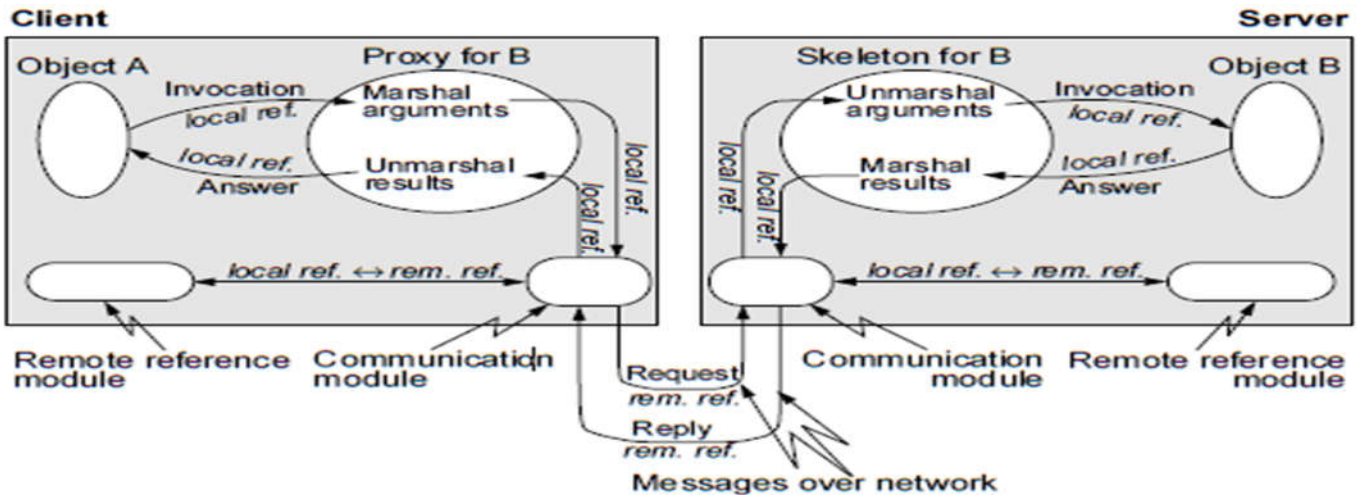

Figure 4.6: Implementation of RMI

Who are the players? Objects

- Object A asks for a service
- Object B delivers the service

Who more?

The proxy for object B

If an object A holds a remote reference to a (remote) object B, there exists a proxy object for B on the machine which hosts A. The proxy is created when the remote object reference is used for the first time. For each method in B there exists a corresponding method in the proxy.

The proxy is the local representative of the remote object ⇒ the remote invocation from A to B is initially handled like a local one from A to the proxy for B.

At invocation, the corresponding proxy method marshals the arguments and builds the message to be sent, as a request, to the server.

After reception of the reply, the proxy *unmarshals* the received message and sends the results, in an answer, to the invoker.

The skeleton for object B

On the server side, there exists a skeleton object corresponding to a class, if an object of that class can be accessed by RMI. For each method in B there exists a corresponding method in the skeleton.

The skeleton receives the request message, unmarshals it and invokes the corresponding method in the remote object; it waits for the result and marshals it into the message to be sent with the reply.

A part of the skeleton is also called ***dispatcher***. The dispatcher receives a request from the *communication module*, identifies the invoked method and directs the request to the corresponding method of the skeleton.

Communication module

The communication modules on the client and server are responsible of carrying out the exchange of messages which implement the request/reply protocol needed to execute the remote invocation.

The particular messages exchanged and the way errors are handled, depends on the RMI semantics which is implemented.

Remote reference module

The remote reference module translates between local and remote object references. The correspondence between them is recorded in a *remote object table*.

Remote object references are initially obtained by a client from a so called *binder* that is part of the global name service (it is not part of the remote reference module). Here servers register their remote objects and clients look up after services.

Given the specification of the server interface and the standard representations, an interface compiler can generate the classes for proxies and skeletons

- ✓ Object A and Object B belong to the application.
- ✓ Remote reference module and communication module belong to the middleware.
- ✓ The proxy for B and the skeleton for B represent the so called *RMI software*. They are situated at the border between middleware and application and usually can be generated automatically with help of available tools that are delivered together with the middleware software.

Basic steps of client and server objects communication in RMI

Step 1. The calling sequence in the client object activates the method in the proxy corresponding to the invoked method in B.

Step 2. The method in the proxy packs the arguments into a message (marshalling) and forwards it to the communication module.

Step 3. Based on the remote reference obtained from the remote reference module, the communication module initiates the request/reply protocol over the network.

Step 4. The communication module on the server's machine receives the request. Based on the local reference received from the remote reference module the corresponding method in the skeleton for B is activated.

Step 5. The skeleton method extracts the arguments from the received message (unmarshalling) and activates the corresponding method in the server object B.

Step 6. After receiving the results from B, the method in the skeleton packs them into the message to be sent back (marshalling) and forwards this message to the communication module.

Step 7. The communication module sends the reply, through the network, to the client's machine.

Step 8. The communication module receives the reply and forwards it to the corresponding method in the proxy.

Step 9. The proxy method extracts the results from the received message (unmarshalling) and forwards

4.7. Message-Oriented Communication

The message-oriented communication does not attempt to hide the fact that communication is taking place. Instead its goal is to make the use of flexible message passing easier. It is based around the model of processes sending messages to each other. Underlying message-oriented communication has two orthogonal properties. Communication can be synchronous or asynchronous, and it can be transient or persistent. Message-oriented communication is provided by message-oriented middleware

(MOM). Besides providing many variations of the send() and receive() primitives, MOM also provides infrastructure required to support persistent communication. The send() and receive() primitives offered by MOM also abstract from the underlying operating system or hardware primitives. As such, MOM allows programmers to use message passing without having to be aware of what platforms their software will run on, and what services those platforms provide. As part of this abstraction MOM also provides marshalling services. MPI (Message Passing Interface) is an example of a MOM that is geared toward high-performance transient message passing. MPI is a message passing library that was designed for parallel computing. It makes use of available networking protocols, and provides a huge array of functions that basically perform synchronous and asynchronous send() and receive().

Another example of MOM is MQ Series from IBM. This is an example of a message queuing system. Its main characteristic is that it provides persistent communication. In a message queuing system, messages are sent to other processes by placing them in queues. The queues hold messages until an intended receiver extracts them from the queue and processes them. Communication in a message queuing system is largely asynchronous. The basic queue interface is very simple. There is a primitive to append a message onto the end of a specified queue, and a primitive to remove the message at the head of a specific queue. These can be blocking or nonblocking. All messages contain the name or address of a destination queue. Messages can only be added to and retrieved from local queues. Senders place messages in source queues (or send queues), while receivers retrieve messages from destination queues (or receive queues). The underlying system is responsible for transferring messages from source queues to destination queues. This can be done simply by fetching messages from source queues and directly sending them to machines responsible for the appropriate destination queues. Or it can be more complicated and involve relaying messages to their destination queues through an overlay network of routers. An example of such a system is shown in Figure 4.7. In the figure, an application on sender A sends a message to an application on receiver B. It places the message in its local source queue, from where it is forwarded through routers R1 and R2 into the receiver's destination queue.

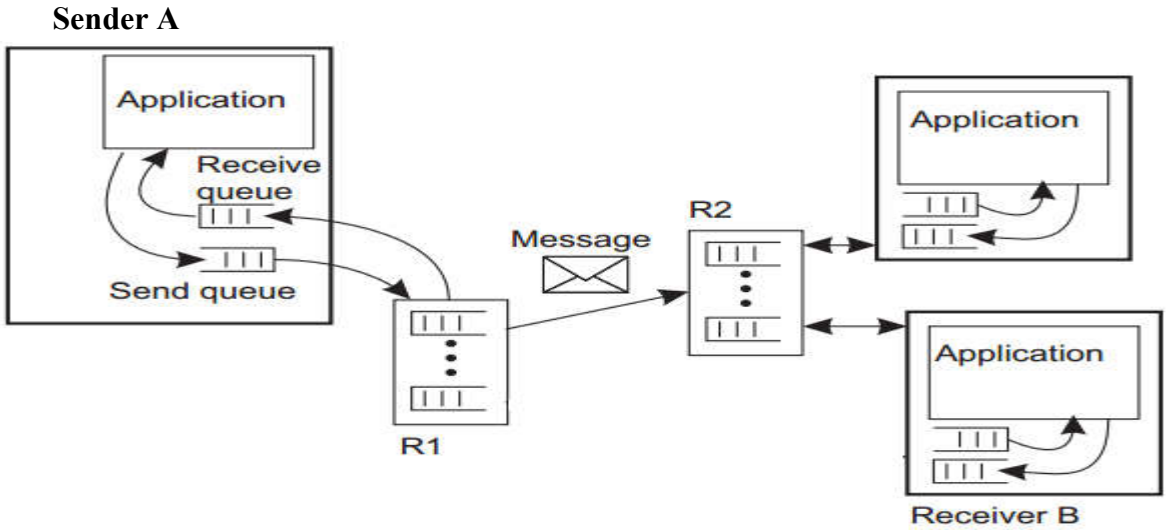

Figure 4.7: An example of a message queuing system.

4.8. Stream-oriented Communication

Whereas the previous middleware frameworks/protocols dealt with discrete communication (that is they communicated chunks of data), the Stream oriented communication deals with continuous communication, and in particular with the sending and receiving of continuous media. In continuous media, data is represented as a single stream of data rather than discrete chunks (for example, an email is a discrete chunk of data; a live radio program is not). The main characteristic of continuous media is that besides a spatial relationship (i.e., the ordering of the data), there is also a temporal relationship between the data. Film is a good example of continuous media. Not only must the frames of a film be played in the right order, they must also be played at the right time, otherwise the result will be incorrect.

A stream is a communication channel that is meant for transferring continuous media. Streams can be set up between two communicating processes, or possibly directly between two devices (e.g., a camera and a TV). Streams of continuous media are examples of isochronous communication that is communication that has minimum and maximum end-to-end time delay requirements. When dealing with isochronous communication, quality of service is an important issue. In this case quality of service is related to the time dependent requirements of the communication. These requirements describe what is required of the underlying distributed system so that the temporal relationships in a stream can be preserved. This generally involves timeliness and reliability.

Further reading assignment

1. Layered protocols/OSI reference layer & TCP/IP model
2. Client-Server TCP
3. RPC vs RMI
4. Challenges with RPC and RMI
5. Group communication
6. How middleware ensures quality of service on stream oriented communication?