Distributed Systems & Computer Networks
From Messages to Bits and Back

by

Amir Abbas & Omar Bashir

1.0 Introduction

With ***distributed computing***, different hardware and software components of an application can be located on different computers connected via networks. Distributed computing allows application developers to spread an application over an entire computer network. It, therefore, provides the users with flexible, scaleable and modular architectures to allow them to access information and share resources available on different workstations and servers which are connected via a network. Moreover, such an architecture may be exploited to improve the availability of these resources by replicating services over different computers (Kramer 1994).

As mentioned earlier, computer networks provide the necessary means of communicating between the components of a distributed system. A ***computer network*** is defined as an interconnected collection of autonomous computers. Two computers are interconnected if they are able to exchange information. An autonomous computer is a system that can operate independently without its operation being affected by other computers (Potter 1985, Tannenbaum 1989). The entire hardware and software facilities required to implement a computer network (i.e. circuits, switches, interfaces, protocol managers and communication handlers etc.) are collectively termed as the ***communication sub-system*** (Coulouris et al 1994).

Distributed systems are generally constructed so as to isolate the communication sub-system from the application components. The application components virtually communicate with each other by passing messages of arbitrarily lengths. The syntax as well as the semantics of these messages are defined in the application level protocols. Physically, the application level components pass these messages to the communication sub-system via well defined interfaces. The communication subsystem actually communicates these messages as sets of packets. Each packet has a pre-defined maximum size and structure to allow efficient communication over the underlying physical network (Figure - 1).

The components constituting the communication subsystem deal with the segmentation of messages into packets and their reassembly, transmission and reception of data packets, error detection, timer control and retransmission. These components neither deal with the semantics of the messages nor do they differentiate between different applications on the basis of their messages. This isolation of the application components from the communication mechanisms allows the implementation of a wide variety of

communication media and protocols without performing any changes in the application software.

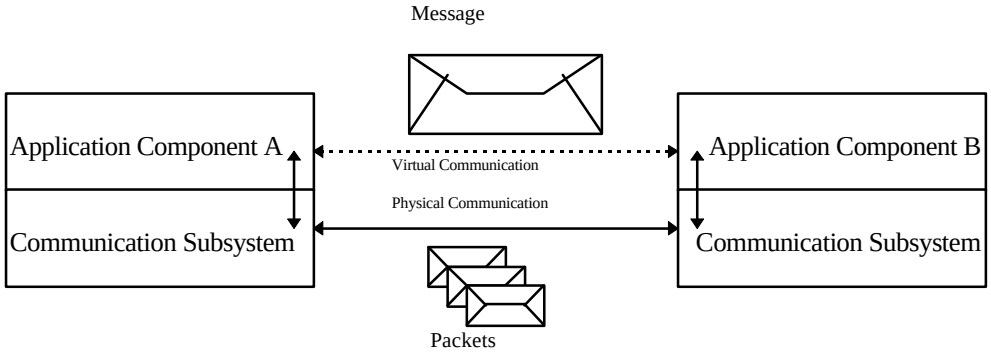

Figure-1 : Data Communication in Distributed Systems

This article provides a brief overview of different aspects of distributed computing. The next section discusses networking concepts. Additionally, the client server model for distributed systems is discussed. Software constructs that enable programmers to develop distributed applications at a higher level of abstraction are discussed before concluding this article.

2.0 Data Communication : OSI Reference Model

In the early 1980s, the International Organization for Standardization (ISO) recognized the need for a network model that would help vendors create interoperable network implementations. The OSI reference model, released in 1984, addresses this need. The OSI reference model is the primary architectural model for intercomputer communications. Although other architectural models (mostly proprietary) have been created, most network vendors relate their network products to the OSI reference model when they want to educate users about their products.

2.1 Hierarchy of OSI Reference Model

The OSI reference model divides the problem of moving information between computers over a network medium into seven smaller and in fact, more manageable problems. Each of these seven smaller problems is reasonably self-contained and therefore more easily solved without excessive reliance on external information.

Each of the seven problem areas is solved by a separate ***layer*** of the model. Most network devices implement all seven layers. To streamline operations, however, some network implementations skip one or more layers. The best example is TCP/IP suite of protocols that does not use presentation and session layers. The lower two OSI layers are implemented with hardware and software; the upper five layers are generally implemented in software.

The OSI reference model describes how information makes its way from application programs (such as word processors) through a network medium (such as wires and fiber optic cables) to another application program in another computer. As the

information to be sent descends through the layers of a given system, it looks less and less like human language and more and more like the ones and zeros that a computer understands.

As an example of OSI-type communication, assume that System A in Figure 2 has information to send to System B. The application program in System A communicates with System A's Layer 7 (the top layer), which communicates with System A's Layer 6, which communicates with System A's Layer 5, and so on until System A's Layer 1 is reached. Layer 1 is concerned with putting information on (and taking information off) the physical network medium. After the information has traversed the physical network medium and been absorbed into System B, it ascends through System B's layers in reverse order (first Layer 1, then Layer 2, and so on) until it finally reaches System B's application program.

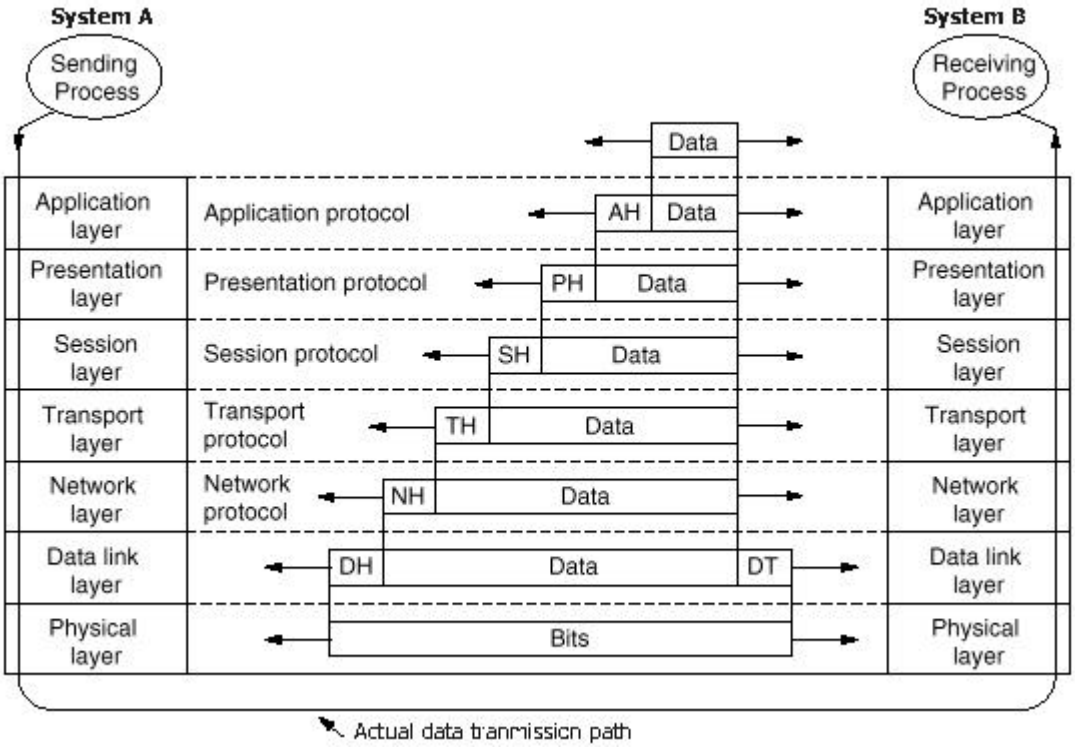

Figure 2: Communication between Two Computer Systems

Although each of System A's layers communicates with its adjacent System A layers, its primary objective is to communicate with its peer layer in System B. That is, the primary objective of Layer 1 in System A is to communicate with Layer 1 in System B; Layer 2 in System A communicates with Layer 2 in System B, and so on. This is necessary because each layer in a system has certain tasks it must perform. To perform these tasks, it must communicate with its peer layer in the other system.

The OSI model's layering precludes direct communication between peer layers in different systems. Each layer in System A must therefore rely on services provided by

adjacent System A layers to help achieve communication with its System B peer. The relationship between adjacent layers in a single system is shown in Figure 3.

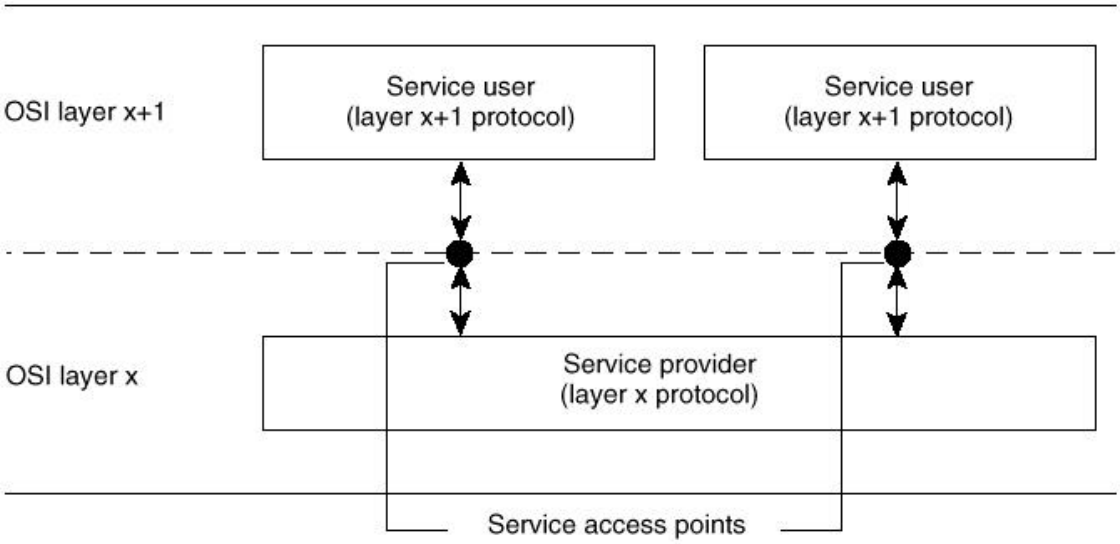

Figure 3: Relationship between Adjacent Layers in a Single System

Assume Layer 4 in System A must communicate with Layer 4 in System B. To do this, Layer 4 in System A must use the services of Layer 3 in System A. Layer 4 is said to be the ***service user***, while Layer 3 is the ***service provider***. Layer 3 services are provided to Layer 4 at a ***service access point*** (SAP), which is simply a location at which Layer 4 can request Layer 3 services. As the figure shows, Layer 3 can provide its services to multiple Layer 4 entities.

2.2 Control Information and Headers

How does Layer 4 in System B know what Layer 4 in System A wants? Layer 4's specific requests are stored as ***control information***, which is passed between peer layers in a block called a ***header*** that is prepended to the actual application information. For example, assume System A wishes to send the following text (called ***data***) to System B:

A quick brown fox jumped over the lazy dog

This text is passed from the application program in System A to System A's top layer. System A's application layer must communicate certain information to System B's application layer, so it prepends that control information (in the form of a coded header) to the actual text to be moved. This information unit is passed to System A's Layer 6, which may prepend its own control information. The information unit grows in size as it descends through the layers until it reaches the network, where the original

text and all associated control information travels to System B, where it is absorbed by System B's Layer 1. System B's Layer 2 strips the Layer 2 header, reads it, and then knows how to process the information unit. The slightly smaller information unit is passed to Layer 3, which strips the Layer 3 header, analyzes the header for actions Layer 3 must take, and so forth. When the information unit finally reaches the application program in System B, it simply contains the original text.

The concept of a header and data is relative, depending on the perspective of the layer currently analyzing the information unit. For example, to Layer 3, an information unit consists of a Layer 3 header and the data that follows. Layer 3's data, however, can potentially contain headers from Layers 4, 5, 6, and 7. Further, Layer 3's header is simply data to Layer 2. This concept is illustrated in Figure 2. Finally, not all layers need to append headers. Some layers simply perform a transformation on the actual data they receive to make the data more or less readable to their adjacent layers. Physical layer generally does not append a header.

2.3 Implementation of OSI Model

The OSI reference model is not a network implementation. Instead, it specifies the functions of each layer. In this way, it is like a model or blueprint for the building of an aircraft. After the blueprint is complete, the aircraft must still be built. Any number of aircraft manufacturing companies can be contracted to do the actual work, just as any number of network vendors can build a protocol implementation from a protocol specification. And, unless the blueprint is extremely comprehensive, aircraft built by different companies using the same blueprint will differ from each other in at least minor ways. At the very least, for example, it is likely that the screws will be in different places.

2.4 OSI Layers

Now that the basic features of the OSI layered approach have been described, each individual OSI layer and its functions can be discussed. Each layer has a predetermined set of functions it must perform for communication to occur.

Application Layer

The application layer is the OSI layer closest to the user. It differs from the other layers in that it does not provide services to any other OSI layer, but rather to application processes lying outside the scope of the OSI model. Examples of such application processes include spreadsheet programs, word-processing programs, banking terminal programs, and so on.

The application layer identifies and establishes the availability of intended communication partners, synchronizes cooperating applications, and establishes agreement on procedures for error recovery and control of data integrity. Also, the

application layer determines whether sufficient resources for the intended communication exist.

Presentation Layer

The presentation layer ensures that information sent by the application layer of one system will be readable by the application layer of another system. If necessary, the presentation layer translates between multiple data representation formats by using a common data representation format.

The presentation layer concerns itself not only with the format and representation of actual user data, but also with data structures used by programs. Therefore, in addition to actual data format transformation (if necessary), the presentation layer negotiates data transfer syntax for the application layer.

Session Layer

As its name implies, the session layer establishes, manages, and terminates sessions between applications. Sessions consist of dialogue between two or more presentation entities (recall that the session layer provides its services to the presentation layer). The session layer synchronizes dialogue between presentation layer entities and manages their data exchange. In addition to basic regulation of conversations (sessions), the session layer offers provisions for data expedition, class of service, and exception reporting of session-layer, presentation-layer, and application-layer problems.

Transport Layer

The boundary between the session layer and the transport layer can be thought of as the boundary between application-layer protocols and lower-layer protocols. Whereas the application, presentation, and session layers are concerned with application issues, the lower four layers are concerned with data transport issues.

The transport layer attempts to provide a data transport service that shields the upper layers from transport implementation details. Specifically, issues such as how reliable transport over an internetwork is accomplished are the concern of the transport layer. In providing reliable service, the transport layer provides mechanisms for the establishment, maintenance, and orderly termination of virtual circuits, transport fault detection and recovery, and information flow control (to prevent one system from overrunning another with data).

Network Layer

The network layer is a complex layer that provides connectivity and path selection between two end systems that may be located on geographically diverse ***subnetworks***.

A subnetwork, in this instance, is essentially a single network cable (sometimes called a ***segment***).

Because a substantial geographic distance and many subnetworks can separate two end systems desiring communication, the network layer is the domain of routing. Routing protocols select optimal paths through the series of interconnected subnetworks. Traditional network layer protocols then move information along these paths.

Link Layer

The link layer (formally referred to as the data link layer) provides reliable transit of data across a physical link. In so doing, the link layer is concerned with ***physical*** (as opposed to ***network***, or ***logical***) addressing, network topology, line discipline (how end systems will use the network link), error notification, ordered delivery of frames, and flow control.

Physical Layer

The physical layer defines the electrical, mechanical, procedural, and functional specifications for activating, maintaining, and deactivating the physical link between end systems. Such characteristics as voltage levels, timing of voltage changes, physical data rates, maximum transmission distances, physical connectors, and other similar attributes are defined by physical layer specifications.

3.0 The Internet Protocol Suite

The Internet Protocol suite, commonly referred to as the TCP/IP protocol suite was developed to be used in the Internet and other internetworking applications. This protocol suite uses a layered model that does not conform precisely to the ISO 7 layer model. The TCP/IP protocol stack includes three internetwork protocol layers that must be supported by the underlying network interface (Figure - 4). The Internet transport layer provides two transport protocols. These are the ***TCP (Transmission Control Protocol)*** and ***UDP (User Datagram Protocol)***. TCP is a reliable connection oriented protocol and UDP is a connectionless datagram protocol that does not guarantee reliable communication. The protocol used by the network layer of the TCP/IP protocol suite (also known as the Internet layer) is known as the ***Internet Protocol (IP)***. The Internet layer provides the basic transmission mechanism for the Internet and other networks using the TCP/IP protocol suite.

The TCP/IP specification does not specify the layer below the IP layer. IP packets in the Internet layer are transformed into packets for transmission over almost any combination of networks or data links. This independence of TCP/IP protocol suite from the underlying transmission technology is the main reason for its success. This allows internetworks to be built from many heterogeneous networks and data links. TCP and UDP protocols provide the users and application programs a single virtual transport mechanism between these

applications. The Internet layer provides a uniform virtual network service to the transport layers on different computers. Each computer on the Internet is identified by a unique 32 bit number known as its IP address. The Internet layer uses these IP addresses to route IP packets to the destination computers. This uniform virtual network provided by the Internet layer hides the diversity of the underlying transmission media from the upper layers.

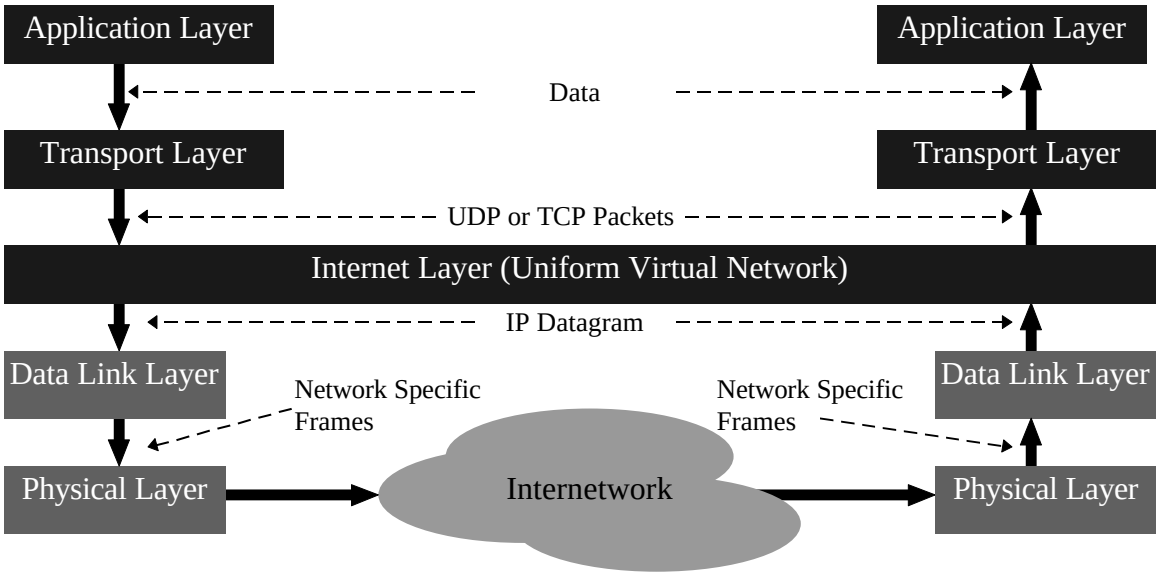

Figure - 4: The Internet Protocol Suite

4.0 The Client Server Model

The client server model is the most widely adopted model for distributed systems. A ***server*** is a manager for one or more resources (hardware or software), e.g., printers, disk drives, databases etc. ***Clients*** are the users of server's resources.

Thus, in a client server model, all the shared resources are held and managed by the server processes. A server process waits passively for the client's request to establish a connection. A connection between a client and a server is established when the server accepts the client's request for the connection. A client can then issue requests to the server to gain access to one of its resources. If the request is valid, the server performs the requested actions and sends a reply to the client process (Figure - 5). These client and server processes may reside on the same computer or on different computers connected via a network.

The interaction between a client and a server is known as the ***request reply protocol***. Generally, this request reply protocol is synchronous. The client process blocks until the reply arrives from the server. It is possible to implement asynchronous request reply communication in situations where the clients can afford to retrieve replies later (Coulouris et al 1994). Thus the exchange consists of

1. transmission of a request from a client process to the server process

2. execution of the request by the server process
3. transmission of a reply to the client

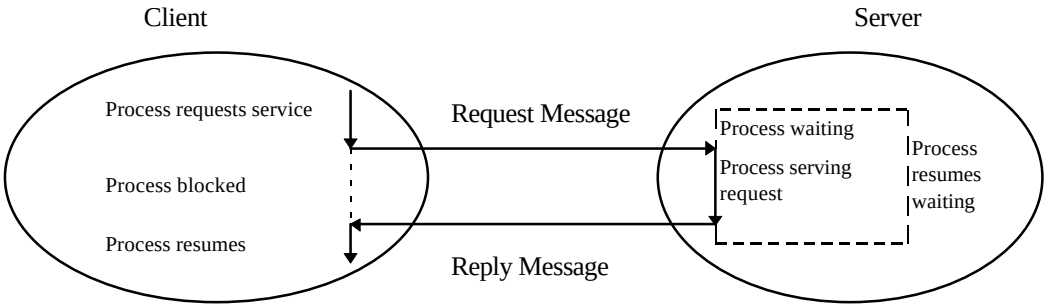

Figure - 5 : Client Server Interaction

However other interactions are also possible. It is possible for the clients to issue a series of requests and the servers to provide a series of responses. In several other cases, the servers start providing information as soon as a connection is established, without waiting for an explicit request for information (Comer 97). More complex interactions are also possible. For example, a client application is not restricted to accessing a single server while performing a specific operation. A client may thus contact a number of different servers for different services. Similarly a client may also access a number of different servers to receive a specific service in an optimal manner. Additionally, servers may perform further client server interactions in which these servers become clients of other servers (Comer 97).

Clients and servers are components of distributed applications and, therefore, use the transport protocol to communicate. Client and server programs interact directly with the transport layer protocol (i.e. the TCP layer). The transport layer protocol then uses the lower layer protocols to send and receive individual messages (Figure - 6). Thus, computers require a complete protocol stack implementation to allow the clients and servers to communicate (Comer 97).

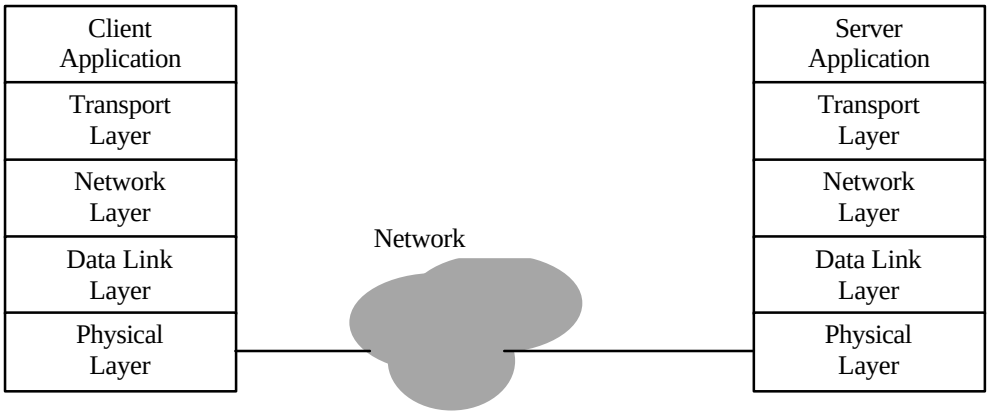

Figure - 6 : Client Server Application and the Protocol Stack

It is possible to execute multiple server applications on the same computer. These server applications utilise the same physical connection to the network. Clients can communicate with any of these servers which use the same physical link and the protocol stack (Figure - 7).

The transport protocol assigns each server a unique identifier, for example, TCP uses 16 bit integer values, known as protocol port numbers, to identify services. Both clients and servers use this identifier. The server needs to register this identifier with the local protocol software at start up. While attempting to connect to a particular server, the client needs to specify this identifier. This identifier is included in the connection request that is sent to the server. The transport protocol at the server machine uses this identifier to identify the server which should handle the request (Comer 97).

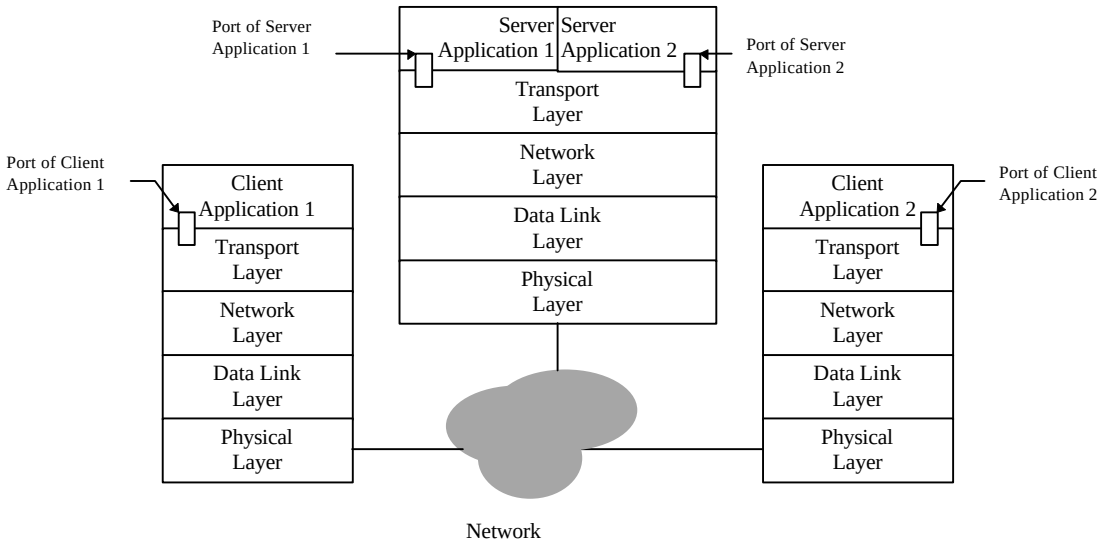

Figure - 7 : Multiple Server Applications on a Single Hardware

A single server, if implemented as a single process, may take substantial time to process a request. Clients may encounter significant waits while earlier requests are satisfied. If the resource is of the type that can be used by a number of clients at a time, significantly improved responsiveness can be obtained by providing multiple parallel server processes, any one of which can satisfy the client's requests. However, implementing idle parallel processes may be wasteful. Thus the number of processes chosen is a tradeoff between responsiveness and waste of resources.

Where the overhead of creating and terminating a server process is negligible relative to the overall processing of a single request, it may be reasonable to have a multiplexer process waiting to accept requests. When the multiplexer process receives a request, it can create a server process to handle that request. As context switching can be quite expensive on a computer's resources, servers may therefore be best implemented as threads within a

single process. As threads share memory and other resources, there is less overhead in switching from one thread to another than in switching from one process to another.

5.0 Programming Abstractions

5.1 Sockets

The Socket API (Application Programming Interface) was originally developed by the University of California at Berkeley to facilitate IPC (Inter-Process Communication) within a computer system or on different computers connected via networks. Socket functions form part of the BSD (Berkeley Software Distribution) Unix operating system and operating systems derived from it. Because of their popularity, vendors of several other systems have decided to include the socket functions in their systems. However, in order not to modify their operating systems, they have developed socket libraries which provide the socket API. These libraries use the same function names, arguments and syntax as the socket API, thereby ensuring source code portability (Comer 97).

A ***socket*** is an end point used by a process for bi-directional communication with a socket associated with another process. Thus data can flow in both directions simultaneously. Sockets can provide connection oriented (reliable) as well as connection-less or datagram (unreliable) communication. Sockets in the Unix operating system are integrated with the I/O. Thus an application communicates through a socket in a manner similar to the way an application transfers data to or from a file, i.e., the ***open-read-write-close*** paradigm is used.

The Windows Sockets specification defines a binary compatible network programming interface for Microsoft Windows. These are based on the Unix sockets implementation in the BSD release 4.3. The specification includes both the BSD style socket routines as well as the extensions specific to Windows. Windows Sockets abstract away the underlying network so that the programmer does not have to be knowledgeable about the network.

The Microsoft Foundation Class (MFC) library supports programming with Windows Sockets API by supplying two classes. One of these classes, ***CSocket***, provides a high level of abstraction to simplify network communication programming. It is derived from the ***CAsyncSocket*** class and inherits its encapsulation of the Windows Sockets API. The CAsyncSocket class is generally used by programmers who have detailed knowledge regarding network communications. As this class encapsulates the Windows Sockets API at a very low level, the programmers have to manage issues related to blocking, byte order and string conversion while using the CAsyncSocket class. Alternatively, the CSocket class provides blocking which is essential for synchronous operations. The blocking functions of the CSocket class, when invoked, wait until the operation completes.

5.2 Remote Procedure Call (RPC)

Communication within a client server environment is largely accomplished either by the message passing or by the remote procedure call mechanisms.

Message passing mechanisms consist of a direct exchange of units of data between the client and server components that are running on different computing systems. In its simplest form, a client passes a message to the server, requesting the execution of a process. The message consists of the type of process to be executed by the server as well as the parameters required by the process to execute. Once the server has carried out the required processing, it sends a set of messages to the client that forms the results of the operation conducted at the server (Figure - 8).

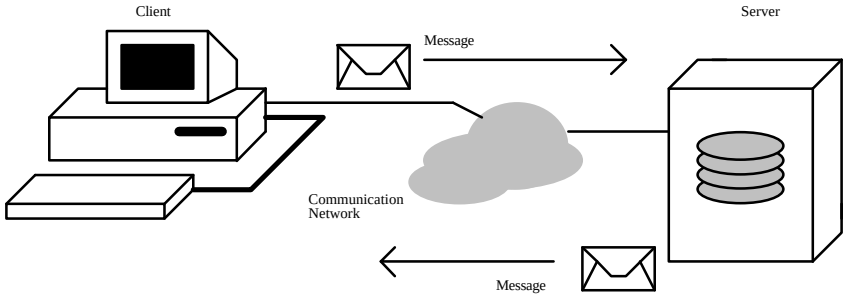

Figure - 8 : Message Passing Mechanism

Remote procedure call (RPC) mechanism extends the familiar procedure call programming paradigm from the local computing system environment to a distributed environment. With an RPC facility, the calling procedure and the called procedure can execute on different computing systems in the network (Figure - 9). RPC mechanism attempts to hide from the application programmer the fact that distribution is taking place.

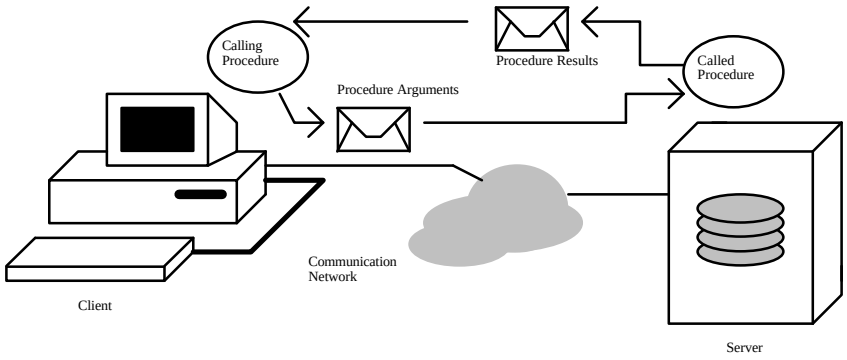

Figure - 9 : Remote Procedure Calling

RPC mechanisms are generally built on top of message passing mechanisms and implement a simple request - response protocol. The calling procedure makes a request for the execution of the called procedure and may pass a set of parameters to the called procedure. The called procedure then executes and may pass a set of results

back to the calling procedure. The process of formatting and communicating messages between the called and the calling procedure is known as ***marshaling***.

RPC has its inherent disadvantages. These include an extended overhead which may be 4 orders of magnitude higher than that of the local call. Moreover, there is no possibility of sharing memory between the calling and the called procedures (Sauer 1993). Use of pointers may also be inconvenient. Finally means of translating references between heterogeneous processing architectures must also be provided.

RPC is usually implemented with connectionless (datagram) protocols such as UDP. Failure of networking devices may therefore inhibit messages to be communicated between the client and the server. An RPC mechanism typically includes a timer mechanism that retries calls or replies which have not been properly acknowledged. There is a possibility that a call may be executed more than once. This situation is generally dealt either by having the called functions to be ***idempotent*** (where multiple calls have the same effect as a single call) or by guaranteeing that any given function is called at most once.

Moreover message passing protocols used in networks generally provide limited length datagrams. These message sizes may not be regarded as adequate for use in transparent RPC systems as the arguments or the results of the procedures may be of sizes larger than these. This problem is generally solved by designing a protocol on top of the message passing operations for passing ***multipacket*** request and reply messages (Coulouris et al 94). Thus the requests and replies that do not fit within a single datagram are transmitted as multiple packets where the message is made of a sequence of datagrams.

5.3 Distributed Object Technology (DOT)

Distributed object technology (DOT) is a combination of object technology, distribution technology and web technology. DOT allows the use of objects in distributed systems to represent units of distribution, movement and communications (Wallanau et al 1997). Objects are integrated using ***Object Request Brokers*** (ORBs).

ORB is a middleware that allows the objects to request each other for services. Objects that ORB works with can function either as clients or servers. If an object receives or processes a request, it is functioning as a server. Alternatively, if the object makes a request, then it is acting as client (Stanek 1997).

ORB intercepts requests from one object to another. Then it locates the object which is supposed to service the request. Once this object has been located, appropriate methods in the receiving object are invoked. Method parameters are passed to the server object and the results are returned to the client object. Operation of ORB is transparent, i.e., client and server objects may be located on a local or remote machines (Stanek 1997).

The ***Object Management Group (OMG)*** defines the ***Common Object Request Broker Architecture (CORBA)***. CORBA provides a standard by which object technology can be used in distributed computing environments. CORBA's strengths include heterogeneity as well as the incorporation and integration of existing protocols and applications.

When using CORBA, interfaces to objects can be defined statically in OMG's ***Interface Definition Language (IDL)***. Once an interface is described in IDL, it is compiled for a particular programming language and host machine. Compilation generates the code required to enable communication between the client and server objects. The client process uses a ***stub*** which acts as a local surrogate for the server process. The stub services the call, takes the parameters, packages them for communication and transmits the package to the remote object.

The server process uses a ***skeleton*** generated by the IDL compiler. The skeleton receives the incoming byte stream and reconstructs the parameters for the method called in the server process. At the end of the call, the skeleton converts the return values to a form that facilitates communication over the network back to the original stub. The stub converts the return values from the network representation to a format required by the client process. These values are then passed to the client process.

Clients may use the ***Dynamic Invocation Interface*** which allows the dynamic construction of object invocations. Instead of calling a stub routine, a client can specify, through a call or a sequence of calls, the object to be invoked, the operation to be performed and the set of parameters. The client can obtain the information regarding specific operations to be performed and the type of parameters to be passed from an ***Interface Repository*** service. This service represents the components of an interface as persistent objects which can be accessed at runtime.

Similarly ***Dynamic Skeleton Interface*** can dynamically handle object invocations. Thus, rather than being accessed through a skeleton specific to a particular operation, an object's implementation is accessed in a manner similar to the client side's Dynamic Invocation Interface. The information of parameters related to the invocation can be dynamically obtained from the Interface Repository.

The implementation code must provide the description of all the operation parameters to the ORB. The ORB provides the values of any input parameters for use in performing the operation. The implementation code provides the return values as well as exceptions to the ORB at the end of the operation. Client requests generated either through a stub or through the Dynamic Invocation Interface can use the Dynamic Skeleton Interface to perform object invocations. Results are identical in both cases.

Figure - 10 provides a simplified description of object invocation between the clients and the server object implementations via the ORB.

Object implementations use an ***object adapter*** to access services provided by the ORB. These services include the generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations and registration of implementations.

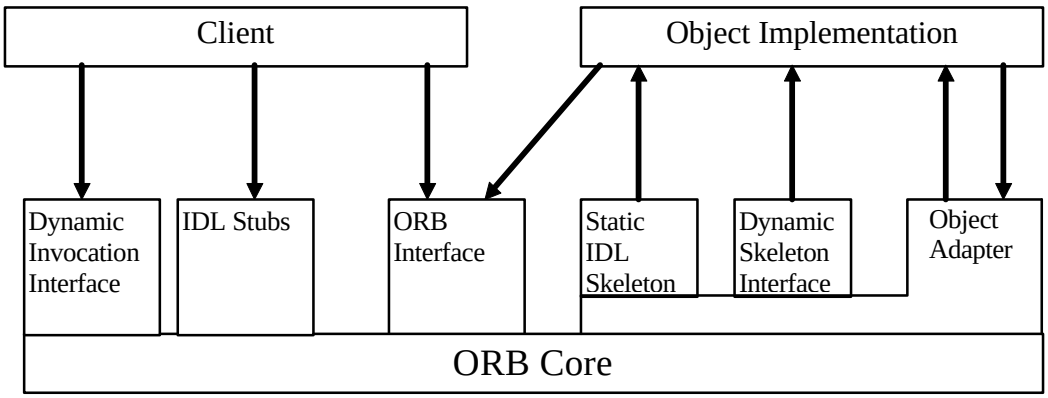

Figure - 10 : Object Invocation using CORBA

Java Version 1.1 provides a ***Remote Method Invocation (RMI)*** feature. This enables a program operating on a client computer to make method calls on an object located on a remote server machine. RMI is Java's native ORB as it supports method invocations on remote objects. A client can invoke the methods of a remote object with the same syntax that it uses to invoke methods on a local object. The RMI API provides classes and methods that handle the underlying communications and parameter referencing requirements for accessing remote objects. RMI also handles serialisation of objects that are passed as arguments to the methods of these remote objects (Heller et al 1997).

RMI consists of 3 layers, i.e., the ***stub/skeleton***, ***remote reference*** and ***transport*** layers (Figure - 11).

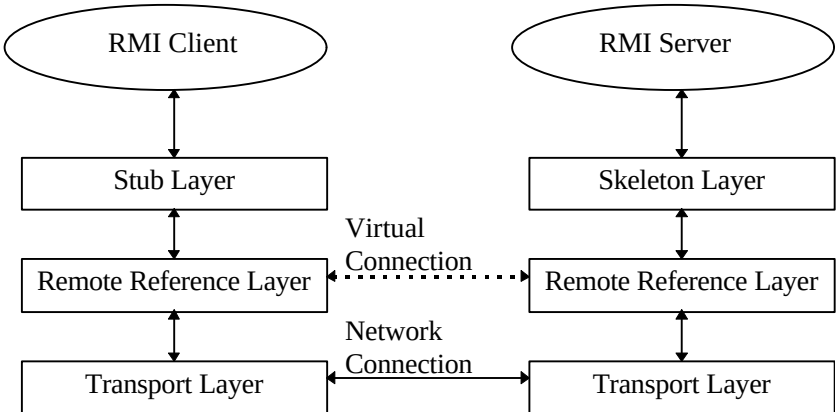

Figure - 11 : Java RMI Architecture

Once a remote method is invoked, the request starts at the top of the stub on the client side. The client references the stub as a proxy for the object on the remote machine. Stubs define all the interfaces that the remote object implementation supports. The program executing on the client machine references the stub as any other local object. The stub maintains a connection to the server side object.

The ***remote reference layer (RRL)*** on the client side returns a marshal stream to the stub. Client side RRL uses this marshal stream to communicate with the server side RRL. The stub serialises the parameter data and passes the serialised data to the marshal stream. After the remote method has been executed, the RRL passes any serialised return values back to the stub which deserialises these return data.

The ***skeleton*** is the server side construct that interfaces with the server side RRL. The skeleton receives method invocation requests. The server side RRL unmarshals any arguments that are sent to the remote method. The skeleton then makes a call to the actual object implementation on the server side. The skeleton is also responsible for receiving any return values from the remote object and marshaling these onto the marshal stream.

RRL maintains an independent reference protocol that is not specific to any stub or skeleton model. This allows for changing the RRL without affecting the other two layers. RRL deals with the lower level transport interface and is responsible for providing a stream to the stub and skeleton layers.

The ***transport layer*** is responsible for creating and maintaining connections between clients and servers. The transport layer sets up the connections, maintains the existing connections and handles the remote objects existing in its address space. The transport layer, upon receiving a request from the client side RRL, locates the RMI server for the remote object that is being requested. The transport layer then establishes a socket connection to the server. The established connection is then passed to the client side RRL and reference to the remote object is added to an internal table.

At this stage the client is considered to be connected to the server. However, the transport layer disconnects the client and the server if there is no activity on the connection for a considerable period of time. This time-out period is 10 minutes.

The above mentioned points also highlight the major difference between CORBA and Java RMI. CORBA is an integration technology as opposed to RMI which is coupled tightly to the Java programming language environment. For a Java client to use RMI to communicate with a remote object implemented in another language, a Java intermediary must be used. Alternatively, CORBA is designed to integrate disparate programming technologies. When two programmers, developing components of an application in two different programming languages, use CORBA to communicate, both the programmers work completely within their respective language environments.

The CORBA ORB presents the stub and skeleton interfaces for the programming languages used to implement client and server respectively. CORBA automatically manages the cross language issues.

6.0 Concluding Remarks

Distributed systems are implemented over a collection of autonomous computers connected by a network. These computers execute software that allows them to interact and coordinate in a manner so as to produce an integrated computing facility. These systems vary in size from a few workstations interconnected by a single LAN, to a WAN interconnecting thousands of heterogeneous processors via different LANs. Distributed applications generally facilitate resource sharing and provide flexible and effective means of communication. Additionally, the availability of computing resources can be improved by replicating these resources and eliminating single points of failure.

Computer networks provide the necessary means for communication between the components of a distributed system. Computer network consist of hardware devices, such as circuits, switches and interfaces, and software components, such as protocol managers and communication handlers. The application level software relies on the network software to provide communication services, without having the application to interact directly with communication devices.

Network software is arranged in a hierarchy of layers, where each layer presents an interface to the layer above it. This allows for the extension and generalisation of some of the properties of the underlying communications system. The software in the lower layers handle most of the low level communication details and problems automatically.

Devices communicating with one another need to follow a well known set of rules and formats. These rules are known as protocols. Protocols specify the sequence of messages that must be exchanged as well as the format of data in these messages. Protocols are not designed in isolation. Rather they are designed and developed in complete cooperative sets called ***protocol suites***.

Each layer in communications software allows a source machine to communicates with the corresponding layer on the destination machine using a separate protocol. Protocol used by a specific layer handles the part of the communications task which is not catered for by other protocols in the specific protocol suite.

To aid application programmers in writing distributed systems, several programming interfaces and libraries of communications procedures have been developed. These facilitate communications using standard protocols at various levels of abstraction. For instance, some APIs and libraries only support message transfer. These messages need to be analysed at their destination to determine the required operation. These messages also include the data required for these operations.

Remote procedure call mechanisms allow client programs to invoke procedures in a server process in a manner similar to the conventional use of procedure calls in high level languages. Recently, significant amount of research and development has been conducted on Object Request Brokers (ORBs) which facilitate the integration of objects distributed across a network.

As these mechanisms automate various common network programming activities, the application developers need only concentrate on message semantics and application logic. This not only increases the productivity of the development team, but also has an impact on the quality of distributed applications they develop.

References

Comer D.E. (1997), *Computer Networks and Internets*, Prentice Hall Inc.

Coulouris G., Dollimore J. & Kindberg T. (1994), *Distributed Systems : Concepts & Design*, Addison Wesley Publishing Co. Inc.

Heller P. & Roberts S. (1997), *Java 1.1 Developers Handbook*, SYBEX Inc.

Kramer J. (1994), *Distributed Software Engineering*, Invited State of the Art Report, Presented at the 16th ICSE Conference, Sorrento, Italy, May 1994.

Potter D. (1985), *Baseband Local Area Networks*, in Data Communications, Networks and Systems, ed. Thomas C. Bartee, Howard W. Sams & Co.

Sauer C. (1993), *Client Server Computing*, in Distributed Computing Environments, edited by David Cerutti & Donna Pierson.

Stanek W.R. (1997), *Distributed Programming with CORBA & IIOP*, PC Magazine, September 23, 1997.

Tannenbaum A.S. (1989), *Computer Networks*, Prentice Hall Inc.

Wallanau K., Weiderman N. & Northrop L. (1997), *Distributed Object Technology with CORBA & Java : Key Concepts and Implications*, Technical Report CMU/SEI-97-TR-004, ESC-TR-97-004, June 1997, Software Engineering Institute, Carnegie Mellon University.