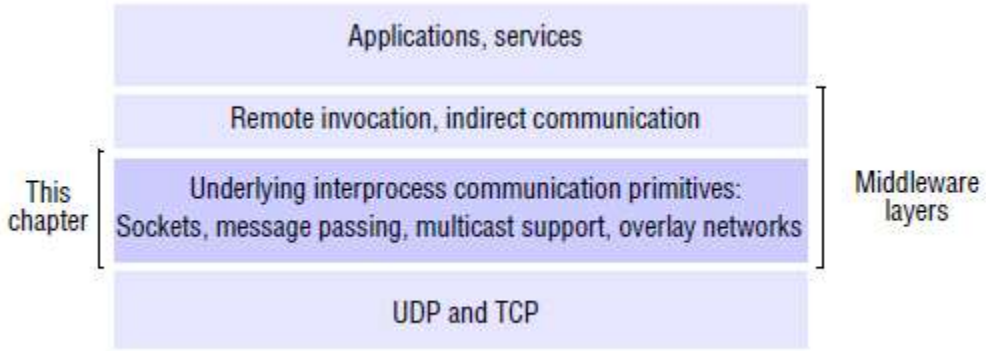
Unit-3
Interprocess Communication

→Introduction: Interprocess communication in the Internet provides both *datagram* and *stream* communication. These concepts are concerned with the communication aspects of middleware. This one is concerned with the design of the components shown in the darker layer in below Figure.



Middleware layers

The application program interface to UDP provides a *message passing* abstraction – the simplest form of interprocess communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called ***datagrams.*** In the Java and UNIX APIs, the sender specifies the destination using a socket – an indirect reference to a particular port used by the destination process at a destination computer.

The application program interface to TCP provides the abstraction of a two-way ***stream*** between pairs of processes. The information communicated consists of a stream of data items with no message boundaries. Streams provide a building block for producer-consumer communication.

A producer and a consumer form a pair of processes in which the role of the producer is to produce data items and the role of the consumer is to consume them. The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used to hold the queued data items is exhausted.

→The API for the Internet protocols:- The general characteristics of interprocess communication are given below. Programmers can use them, either by means of UDP messages or through TCP streams.

1. **The characteristics of interprocess communication:** Message passing between a pair of processes can be supported by two message communication operations, ***send*** and ***receive***, defined in terms of destinations and messages. To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves

the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes.

***#Synchronous and asynchronous communication*:** A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues.

Communication between the sending and receiving processes may be either synchronous or asynchronous. In the ***synchronous* form** of communication, the sending and receiving processes synchronize at every message. In this case, both ***send*** and ***receive*** are ***blocking*** operations. Whenever a *send* is issued the sending process (or thread) is blocked until the corresponding *receive* is issued. Whenever *receive* is issued by a process (or thread), it blocks until a message arrives.

In the ***asynchronous* form** of communication, the use of the *send* operation is *non- blocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The ***receive*** operation can have blocking and non-blocking variants. **In the non-blocking variant,** the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

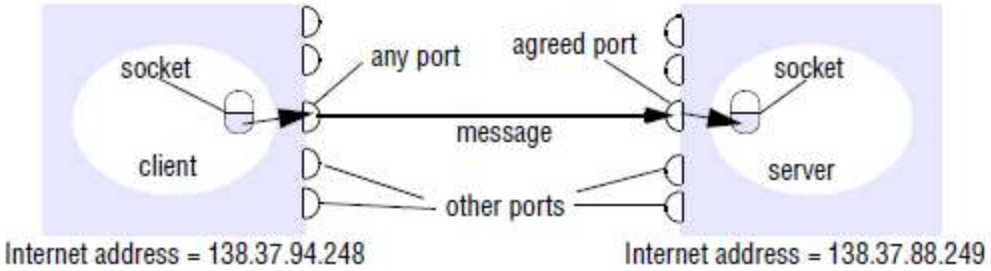
In a system environment such as Java, which supports multiple threads in a single process, the **blocking *receive*** has no disadvantages. **Non-blocking communication** appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control.

#Message destinations*:** Messages are sent to (Internet address, local port***) pairs. A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but can have many senders. Processes may use multiple ports to receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients.

#Reliability: Reliable communication can be defined in terms of ***validity*** and ***integrity.*** A point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a 'reasonable' number of packets being dropped or lost. In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

#ordering: Some applications require that messages to be delivered in ***sender order*** – that is, the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

2. Sockets: Both forms of communication (UDP and TCP) use the ***socket*** abstraction, which provides an endpoint for communication between processes. Sockets originate from BSD UNIX but are also present in most other versions of UNIX, including Linux as well as Windows and the Macintosh OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process(below figure).



Sockets and Ports

For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number (2^{16}) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer.

Java API for Internet addresses*:** As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, ***InetAddress*,** that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames .For example, instances of ***InetAddress that contain Internet addresses can be created by calling a static method of ***InetAddress***, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is ***tecnrt.org***, use:

*InetAddress aComputer = InetAddress.getByName("**tecnrt.org**");*

This method can throw an *UnknownHostException*

3. UDP datagram communication: A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process ***sends*** it and another ***receives*** it. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a local port. A server will bind its socket to a ***server port*** – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.
The following are some issues relating to datagram communication:

***#Message size*:** The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of up to 2^{16} bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size.

#Blocking*:** Sockets normally provide non-blocking ***sends and blocking ***receives*** for datagram communication. The ***send*** operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination. On arrival, the message is placed in a queue for the socket that is bound to the destination port. The message can be collected from the queue by an invocation of *receive* on that socket. Messages are discarded at the destination if no process has a socket bound to the destination port. The method ***receive*** blocks until a datagram is received, unless a timeout has been set on the socket. If the process that invokes the ***receive*** method has other work to do while waiting for the message, it should arrange to use a separate thread.

#Timeouts: The ***receive*** that blocks forever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a ***receive*** operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost. To allow for such requirements, timeouts can be set on sockets.

#Receive from any: The ***receive*** method does not specify an origin for messages. Instead, an invocation of ***receive*** gets a message addressed to its socket from any origin. The ***receive*** method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from.

Failure model for UDP datagrams: Reliable communication can be defined in terms of two properties: ***integrity*** and ***validity.*** The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted.
UDP datagrams suffer from the following failures:

> ***Omission failures***: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination.
> ***Ordering***: Messages can sometimes be delivered out of sender order.

Use of UDP : Some applications can use a service that exhibits occasional omission failures. Example, Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP. UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

- the need to store state information at the source and destination;
- the transmission of extra messages;
- latency for the sender.

Java API for UDP datagrams : The Java API provides datagram communication by means of two classes: ***DatagramPacket*** and ***DatagramSocket***.

->DatagramPacket: This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the Internet address and local port number of the destination socket, as follows:

array of bytes containing message	length of message	Internet address	port number

An instance of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it. This class provides another constructor for use when receiving a message. Its arguments specify an array of bytes in which to receive the message and the length of the array. A received message is put in the *DatagramPacket* together with its length and the Internet address and port of the sending socket. The message can be retrieved from the *DatagramPacket* by means of the method ***getData.*** The methods ***getPort*** and ***getAddress*** access the port and Internet address.

->DatagramSocket: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port.

The class ***DatagramSocket*** provides methods that include the following:

- ***send*** and ***receive***: These methods are for transmitting datagrams between a pair of sockets. The argument of *send* is an instance of ***DatagramPacket*** containing a message and its destination. The argument of *receive* is an empty **DatagramPacket** in which to put the message, its length and its origin. These methods can throw *IOExceptions*.
- ***setSoTimeout***: This method allows a timeout to be set. With a timeout set, the receive method will block for the time specified and then throw an *InterruptedIOException.*
- **connect:** This method is used for connecting to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
     // args give message contents and server hostname
     DatagramSocket aSocket = null;
     try {
          aSocket = new DatagramSocket();
          byte [] m = args[0].getBytes();
          InetAddress aHost = InetAddress.getByName(args[1]);
          int serverPort = 6789;
          DatagramPacket request =
                new DatagramPacket(m,  m.length(), aHost, serverPort);
          aSocket.send(request);
          byte[] buffer = new byte[1000];
          DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
          aSocket.receive(reply);
          System.out.println("Reply: " + new String(reply.getData()));
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e){System.out.println("IO: " + e.getMessage());
       } finally { if(aSocket != null) aSocket.close();}
   }
}
```

UDP client sends a message to the server and gets a reply

Above code shows the program for a client that creates a socket, sends a message to a server at port 6789 and then waits to receive a reply. The arguments of the *main* method supply a message and the DNS hostname of the server. The message

is converted to an array of bytes, and the DNS hostname is converted to an Internet address.

```
import java.net.*;
import java.io.*;
public class UDPServer{
   public static void main(String args[]){
     DatagramSocket aSocket = null;
     try{
          aSocket = new DatagramSocket(6789);
          byte[] buffer = new byte[1000];
          while(true){
            DatagramPacket request = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(request);
            DatagramPacket reply = new DatagramPacket(request.getData(),
                 request.getLength(), request.getAddress(), request.getPort());
            aSocket.send(reply);
          }
     } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
     } catch (IOException e) {System.out.println("IO: " + e.getMessage());
     } finally {if (aSocket != null) aSocket.close();}
   }
}
```

UDP server repeatedly receives a request and sends it back to the client

Above code shows the program for the corresponding server, which creates a socket bound to its server port (6789) and then repeatedly waits to receive a request message from a client, to which it replies by sending back the same message.

4. TCP stream communication: The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read.
The following characteristics of the network are hidden by the stream abstraction:

***#Message sizes*:** The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested.

***#Lost messages*:** The TCP protocol uses an acknowledgement scheme. The sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message.

***#Flow control*:** The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.

***#Message duplication and ordering*:** Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

#Message destinations*:** A pair of communicating processes establishes a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a ***connect request from client to server followed by an ***accept*** request from server to client before any communication can take place.

When pair of processes are establishing a connection, one of them plays the client role and the other plays the server role. The client role involves creating a stream socket bound to any port and then making a ***connect*** request asking for a connection to a server at its server port. The server role involves creating a listening socket bound to a server port and waiting for clients to request connections. The listening socket maintains a queue of incoming connection requests. In the socket model, when the server ***accept*s** a connection, a new stream socket is created for the server to communicate with a client, meanwhile retaining its socket at the server port for listening for ***connect*** requests from other clients.

The pair of sockets in the client and server is connected by a pair of streams, one in each direction. Thus each socket has an *input* stream and an *output* stream. One of the pair of processes can send information to the other by writing to its output stream, and the other process obtains the information by reading from its input stream.

When an application ***closes*** a socket, this indicates that it will not write any more data to its output stream. Any data in the output buffer is sent to the other end of the stream and put in the queue at the destination socket, with an indication that the stream is broken. The process at the destination can read the data in the queue, but any further reads after the queue is empty will result in an indication of end of stream.

The following are some issues related to stream communication:

#Matching of data items: Two communicating processes need to agree as to the contents of the data transmitted over a stream. Example, if one process writes an *int* followed by *double* to a stream, then the reader at the other end must read an i*nt* followed by a *double*.

#Blocking: The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from a input channel, it will block until data becomes available.

#Threads: When a server accepts a connection, it creates a new thread to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients.

Failure model: To satisfy the integrity property of reliable communication, TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets. For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets.

If the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:

- The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection.
- The communicating processes cannot tell whether the messages they have sent recently have been received or not.

Use of TCP : Many frequently used services run over TCP connections. These include the following:

HTTP: The Hypertext Transfer Protocol is used for communication between web browsers and web servers

FTP: The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

***Telnet*:** Telnet provides access by means of a terminal session to a remote computer.

***SMTP*:** The Simple Mail Transfer Protocol is used to send mail between computers.

Java API for TCP streams: The Java interface to TCP streams is provided in the classes: **ServerSocket** and **Socket**:

->***Server Socket****:* This class is intended for use by a server to create a socket at a server port for listening for ***connect*** requests from clients. Its ***accept*** method gets a *connect* request from the queue or, if the queue is empty, blocks until one arrives.

->***Socket*:** This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also ***connects*** it to the specified remote computer and port number.

The *Socket* class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket. Our example uses *DataInputStream* and *DataOutputStream*, which allow binary representations of primitive data types to be read and written in a machine-independent manner.

Below figure shows a client program in which the arguments of the *main* method supply a message and the DNS hostname of the server. The client creates a socket bound to the hostname and server port 7896. It makes a *DataInputStream* and a *DataOutputStream* from the socket's input and output streams, then writes the message to its output stream and waits to read a reply from its input stream.

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
      // arguments supply message and hostname of destination
      Socket s = null;
      try{
        int serverPort = 7896;
        s = new Socket(args[1], serverPort);
        DataInputStream in = new DataInputStream( s.getInputStream());
        DataOutputStream out =
            new DataOutputStream( s.getOutputStream());
        out.writeUTF(args[0]);
        String data = in.readUTF();
        System.out.println("Received: "+ data) ;
      }catch (UnknownHostException e){
        System.out.println("Sock:"+e.getMessage());
      } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
      } catch (IOException e){System.out.println("IO:"+e.getMessage());
      } finally {if(s!=null) try {s.close();}catch (IOException e){/*close failed*/}}
    }
}
```

TCP client makes connection to server, sends request and receives reply

The server program in below Figure opens a server socket on its server port (7896) and listens for ***connect*** requests. When one arrives, it makes a new thread in which to communicate with the client. The new thread creates a *DataInputStream* and a *DataOutputStream* from its socket's input and output streams and then waits to read a message and write the same one back.

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
      try{
        int serverPort = 7896;
        ServerSocket listenSocket = new ServerSocket(serverPort);
        while(true) {
          Socket clientSocket = listenSocket.accept();
          Connection c = new Connection(clientSocket);
        }
      } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
  }
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
      try {
        clientSocket = aClientSocket;
        in = new DataInputStream( clientSocket.getInputStream());
        out =new DataOutputStream( clientSocket.getOutputStream());
        this.start();
      } catch(IOException e) {System.out.println("Connection:"+e.getMessage());
    }
    public void run(){
      try {                    // an echo server
        String data = in.readUTF();
        out.writeUTF(data);
      } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
      } catch(IOException e) {System.out.println("IO:"+e.getMessage());
     } finally { try {clientSocket.close();}catch (IOException e){/*close failed*/}}
    }
}
```

TCP server makes a connection for each client and then echoes the client's request

→External data representation and marshalling:-

The information stored in running programs is represented as data structures whereas the information in messages consists of sequence of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the

same order. The representation of floating-point numbers also differs between architectures.

There are two variants for the ordering of integers: the so-called ***big-endian*** order, in which the most significant byte comes first; and ***little-endian*** order, in which it comes last. Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character.

One of the following methods can be used to enable any two computers to exchange binary data values:

• The values are converted to an agreed external format before transmission and converted to the local form on receipt;

• The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

An agreed standard for the representation of data structures and primitive values is called an ***external data representation***.

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. ***Unmarshalling*** is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus **marshalling** consists of the translation of structured data items and primitive values into an external data representation. Similarly, **unmarshalling** consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

Three alternative approaches for external data representation and marshalling are:

→**CORBA's CDR(**common data representation), which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA.

→**Java's object serialization**, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

→**XML (Extensible Markup Language)**, which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data.

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement of the application programmer. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments.

In the first two approaches, the primitive data types are marshaled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation.

Another issue with regard to the design of marshalling methods is whether the marshaled data should include information concerning the type of its contents. For example, CORBA's representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called *namespaces*.

Two other techniques for external data representation are worthy of mention. Google uses an approach called ***protocol buffers*** to capture representations of both stored and transmitted data. There is also considerable interest **in JSON (JavaScript Object Notation)** an approach to external data representation.

1)CORBA's Common Data Representation (CDR):-

CORBA CDR is the external data representation defined with CORBA 2.0. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types shown below.

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

CORBA CDR for constructed types

- ***Primitive types*:** CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender's ordering, which is specified in each message. The recipient translates if it requires a different ordering. For example, a 16-bit *short* occupies two bytes in the message, and for big-endian ordering, the most significant bits occupy the first byte and the least significant bits occupy the second byte. Each primitive value is placed at an index in the sequence of bytes according to its size.

- ***Constructed types*:** The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure above.

Below figure shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are ***string*, *string* and *unsigned long*.**

index in sequence of bytes	← 4 bytes →	notes on representation
0–3	5	*length of string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1984	*unsigned long*

CORBA CDR message(Person struct with value: {'Smith', 'London', 1984})

Another example of an external data representation is the Sun XDR standard, which is specified in RFC 1832 It was developed by Sun for use in the messages exchanged between clients and servers in Sun.

The type of a data item is not given with the data representation in the message in either the CORBA CDR or the Sun XDR standard. This is because it is assumed that the sender and recipient have common knowledge of the order and types of the data items in a message

***Marshalling in CORBA*:** Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL, which provides a notation for describing the types of the arguments and results of RMI methods. For example, we might use CORBA IDL to describe the data structure in the message in above Figure as follows:

```
struct Person{
    string name;
    string place;
unsigned long year;
    };
```

2)Java Object serialization:- In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations. An object is an instance of a Java class. For example, the Java class equivalent to the *Person struct* defined in CORBA IDL might be:

```
public class Person implements Serializable
        {
                private String name;
                private String place;
                private int year;
                public Person(String aName, String aPlace, int aYear)
                {
                        name = aName;
                        place = aPlace;
                        year = aYear;
                }
                // followed by methods for accessing the instance variables
        }
```

The above class states that it implements the ***Serializable*** interface, which has no methods.

In Java, the term ***serialization*** refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for transmitting in a message. ***Deserialization*** consists of restoring the state of an object or a set of objects from their serialized form.

It is assumed that the process that does the ***deserialization*** has no prior knowledge of the types of the objects in the serialized form. Therefore some information about the class of each object is included in the serialized form. This information enables the recipient to load the appropriate class when an object is deserialized.

Java objects can contain references to other objects. When an object is serialized, all the objects that it references are serialized together with it to ensure that when the object is reconstructed. References are serialized as ***handles***. The handle is a reference to an object within the serialized form.

To serialize an object, its class information is written out, followed by the types and names of its instance variables. If the instance variables belong to new classes, then their class information must also be written out, followed by the types and names of their instance variables. This recursive procedure continues until the class information and types and names of the instance variables of all of the necessary classes have been written out.

The contents of the instance variables that are primitive types, such as integers, chars, booleans, bytes and longs, are written in a portable binary format using methods of the ***ObjectOutputStream*** class. Strings and characters are written by its ***writeUTF*** method using the Universal Transfer Format (UTF-8), which enables ASCII characters to be represented unchanged (in one byte), whereas Unicode characters are represented by multiple bytes. Strings are preceded by the number of bytes they occupy in the stream.

As an example, consider the serialization of the following object:

Person p = new Person("Smith", "London", 1984);

The serialized form is illustrated in Figure below

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name	java.lang.String place	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

Indication of Java serialized form(h0,h1 are handles)

The first instance variable(1984) is an integer that has a fixed length; the second and third variables are strings and are preceded by their lengths.

To *serialize* an object, create an instance of the class ***ObjectOutputStream*** and invoke its ***writeObject*** method, passing the object as its argument. To *deserialize* an object from a stream of data, open an ***ObjectInputStream*** on the stream and use its ***readObject*** method to reconstruct the original object.

The use of reflection*:** The Java language supports ***reflection – the ability to enquire about the properties of a class, such as the names and types of its instance variables and methods. Reflection makes it possible to do serialization and deserialization in a completely generic manner.

Java object serialization uses ***reflection*** to find out the class name of the object to be serialized and the names, types and values of its instance variables.

3)Extensible Markup Language (XML):-

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term ***markup language*** refers to a textual encoding that represents both a text and details as to its structure or its appearance.

Both XML and HTML were derived from **SGML (Standardized Generalized Markup Language).** HTML was designed for defining the appearance of web pages. XML was designed for writing structured documents for the Web. XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. That is, in XML, the tags relate to the structure of the text that they enclose.

XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them.

XML was intended to be used by multiple applications for different purposes. The provision of tags, together with the use of namespaces to define the meaning of the tags, has made this possible. XML documents, being textual, can be read by humans. In practice, most XML documents are generated and read by XML processing software, but the ability to read XML can be useful when things go wrong.

➢**XML elements and attributes** : XML definition of the ***Person*** Structure is given as follows:

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

XML definition of the Person structure

Elements*:** An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in ***Person given above consists of the data ***Smith*** contained within the <*name*> ... </*name*> tag pair. The element with the <*name*> tag is enclosed in the element with the <*person id="123456789"*> ...</*person* > tag pair. The ability of an element to enclose another element allows hierarchic data to be represented.

Attributes: A start tag may optionally include pairs of associated attribute names and values such as *id="123456789",* as shown above. The syntax is the same as for HTML,

in which an attribute name is followed by an equal sign and an attribute value in quotes. Multiple attribute values are separated by spaces.

***Names**:* The names of tags and attributes in XML start with a letter, but can also start with underline or colon. The names can continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case sensitive.

Binary Data: All of the information in XML elements must be expressed as character data. The encrypted elements can be represented in base64 notation which uses alphanumeric characters together with +,/and =.

- **Parsing and well-formed documents:** An XML document must be well formed – that is, it must conform to rules about its structure. A basic rule is that every start tag has a matching end tag. Another basic rule is that all tags are correctly nested – for example, <*x*>..<*y*>..</*y*>..</*x*> is correct, whereas <*x*>..<*y*>..</*x*>..</*y*> is not. Finally, every xml document must have a single root element that encloses all the other elements. When a parser reads an XML document that is not well formed, it will report a fatal error.

  CDATA: XML parsers normally parse the contents of elements because they may contain further nested structures. If text needs to contain an angle bracket or a quote, it may be represented in a special way: for example, *&lt* represents the opening angle bracket. For example, if a place name is to include an apostrophe, then it could be specified in either of the two following ways:

  <place> King&apos Cross </place >
  <place> <![CDATA [King's Cross]]></place >

  XML prolog: Every XML document must have a *prolog* as its first line. The prolog must at least specify the version of XML in use (which is currently 1.0). For example:

  <?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>

  The term ***encoding*** refers to the set of codes used to represent characters – ASCII being the best-known example. The attribute standalone is used to state that whether the document stands alone or is dependent on other documents.

- **XML namespaces :**An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any element that makes use of an XML namespace can specify that namespace as an attribute called ***xmlns***, whose value is a URL referring to the file containing the namespace definitions. For example:

  xmlns:pers = http://www.cdk5.net/person

  The name after ***xmlns***, in this case ***pers*** can be used as a prefix to refer to the elements in a particular namespace, as shown below. The *pers* prefix is bound to *http://www.cdk4.net/person* for the ***person*** element. An XML document may be defined in terms of several different namespaces, each of which is referenced by a unique prefix.

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1984 </pers:year>
</person>
```

Illustration of the use of a namespace in the Person structure

- **XML schemas :** An XML schema defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text. For each element, it defines the type and default value. Below Figure gives an example of a schema that defines the data types and structure of the XML definition of the ***person*** structure.

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name = "name"  type="xs:string"/>
                <xsd:element name = "place"  type="xs:string"/>
                <xsd:element name = "year"  type="xs:positiveInteger"/>
            </xsd:sequence>
            <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
        </xsd:complexType>
</xsd:schema>
```

An XML schema for the Person structure

- **APIs for accessing XML:** XML parsers and generators are available for most commonly used programming languages. For example, there is Java software for writing out Java objects as XML (marshalling) and for creating Java objects from such structures (unmarshalling). Similar software is available in Python for Python data types and objects.

4) Remote object references: When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked.

A ***remote object reference*** is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked.

Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.

There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of

its host computer and the port number of the process that created it with the time of its creation and a local object number.

The local object number is incremented each time an object is created in that process. The port number and time together produce a unique process identifier on that computer. With this approach, remote object references might be represented with a format such as that shown in below figure.

Representation of a remote object reference

In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as the address of the remote object.

→Multicast communication:-

The pairwise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary. A ***multicast operation*** is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. The simplest multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1. ***Fault tolerance based on replicated services***: A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.

2. ***Discovering services in spontaneous networking:*** Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.

3. ***Better performance through replicated data***: Data are replicated to increase the performance of a service. Each time the data changes, the new value is multicast to the processes managing the replicas.

4. ***Propagation of event notifications***: Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications.

IP multicast – An implementation of multicastcommunication:-

- **IP multicast :***IP multicast* is built on top of the Internet Protocol (IP). IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A ***multicast group*** is specified by a Class D Internet address – that is, an address whose first 4 bits are 1110 in IPv4.

The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send datagrams to a multicast group without being a member.

At the application programming level, IP multicast is available only via UDP. An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group. When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number.

The following details are specific to IPv4:

Multicast routers: IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagrams to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called the *time to live*, or TTL for short.

Multicast address allocation: Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). The management of this address space is reviewed annually, with current practice documented in RPC .This document defines a partitioning of this address space into a number of blocks, including:

- Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
- Internet Control Block (224.0.1.0 to 224.0.1.225).
- Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
- Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic.

Multicast addresses may be permanent or temporary. Permanent groups exist even when there are no members – their addresses are assigned by IANA. For example, 224.0.1.1 in the Internet block is reserved for the Network Time Protocol (NTP).

The remainder of the multicast addresses are available for use by temporary groups, which must be created before use and cease to exist when all the members have left. When a temporary group is created, it requires a free multicast address to avoid participation in an existing group. Simple solutions are possible- RFC2908 describes a multicast address allocation architecture (MALLOC) that allocates unique addresses.

A client-server solution is adopted where clients request a multicast address from a multicast address allocation server (MAAS) that ensures allocations are unique.

- **Failure model for multicast datagrams :**Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. This can be called ***unreliable*** multicast, because it does not guarantee that a message will be delivered to any member of a group.

- **Java API to IP multicast:**

The Java API provides a datagram interface to IP multicast through the class ***MulticastSocket*****,** which is a subclass of ***DatagramSocket*** with the additional capability of being able to join multicast groups.

The class ***MulticastSocket*** provides two alternative constructors, allowing sockets to be created to use either a specified local port(6789,above figure) or any free local port.

A process can join a multicast group with a given multicast address by invoking the ***joinGroup*** method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port.

A process can leave a specified group by invoking the ***leaveGroup*** method of its multicast socket.

This can be given in the following program:

The arguments of the *main* method specify a message to be multicast and the multicast address of a group.

After joining that multicast group, the process makes an instance of ***DatagramPacket*** containing the message and sends it through its multicast socket to the multicast group address at port 6789.

After that, it attempts to receive three multicast messages from its peers via its socket, which also belongs to the group on the same port.

When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.

The Java API allows the TTL to be set for a multicast socket by means of the ***setTimeToLive*** method.

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
      // args give message contents & destination multicast group (e.g. "228.5.6.7")
       MulticastSocket s =null;
       try {
          InetAddress group = InetAddress.getByName(args[1]);
          s = new MulticastSocket(6789);
          s.joinGroup(group);
          byte [] m = args[0].getBytes();
          DatagramPacket messageOut =
               new DatagramPacket(m, m.length, group, 6789);
          s.send(messageOut);
          byte[] buffer = new byte[1000];
          for(int i=0; i< 3; i++) {    // get messages from others in group
               DatagramPacket messageIn =
                    new DatagramPacket(buffer, buffer.length);
               s.receive(messageIn);
               System.out.println("Received:" + new String(messageIn.getData()));
          }
          s.leaveGroup(group);
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e){System.out.println("IO: " + e.getMessage());
       } finally { if(s != null) s.close();}
    }
}
```

Multicast peer joins a group and sends and receives datagrams

- **Reliability and ordering of multicast**: A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message. Also, when a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.

  Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so.

  Another isssure is ordering. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members.

Some examples of the effects of reliability and ordering:-

1. ***Fault tolerance based on replicated services***: Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others.

2. ***Discovering services in spontaneous networking***: One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services.

3. ***Better performance through replicated data***: Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.

4. ***Propagation of event notifications***: The particular application determines the qualities required of multicast.

These examples suggest that some applications require a multicast protocol that is more reliable than IP multicast. There is a need for reliable multicast.