UNIT-II

UNIT-II:

Interprocess Communication: Introduction, The API for the Internet Protocols- The Characteristics of Interprocess communication, Sockets, UDP Datagram Communication, TCP Stream Communication; External Data Representation and Marshalling; Client Server Communication; Group Communication- IP Multicast- an implementation of group communication, Reliability and Ordering of Multicast.

Introduction to Interprocess Communication (IPC)

In software systems it is often the case that different processes need to communicate with one another in order to cooperate on some task. This is known as Interprocess Communication (IPC), and over the years many different mechanisms have been employed in order to perform IPC.

Android provides its own mechanisms for IPC, but it should be noted here that the name IPC is a bit of a misnomer on Android, as the Android IPC mechanisms are designed to operate at the level of components rather than processes. In other words a component in one app may use IPC to communicate with a component in another app, or with another component in the same app. Thus IPC in Android is a bit more general than the name would imply.

Asynchronous and Synchronous IPC
In general IPC can take one of two forms:

1. **Asynchronous:** with asynchronous IPC the sender does not wait for the receiver to receive the message. The send function or method returns immediately, and then sometime later the receiver receives the message. If the message requires a response from the receiver, then the message may supply details of how the receiver can send a reply to the sender.

2. **Synchronous:** with synchronous IPC the send and receive are synchronised, and often from the receiver's perspective it appears like a normal method or function call, with any response from the receiver passed to the sender as the return value from the send function or method. With synchronous IPC, if the receiver is slow responding to the message the sender will block until the receiver responds. This is often undesirable in user interface components.

Android provides both asynchronous and synchronous IPC.

The API for the Internet Protocols:-

An application program interface (**API**) is a set of routines, protocols, and tools for building software applications. Basically, **an API** specifies how software components should interact. Additionally, **APIs** are used when programming graphical user interface (GUI) components.

Internet Protocols:-

Transmission Control Protocol (TCP)

TCP is a connection oriented protocol and offers end-to-end packet delivery. It acts as back bone for connection.It exhibits the following key features:

- Transmission Control Protocol (TCP) corresponds to the Transport Layer of OSI Model.
- TCP is a reliable and connection oriented protocol.
- TCP offers:
    - Stream Data Transfer.
    - Reliability.
    - Efficient Flow Control
    - Full-duplex operation.
    - Multiplexing.
- TCP offers connection oriented end-to-end packet delivery.
- TCP ensures reliability by sequencing bytes with a forwarding acknowledgement number that indicates to the destination the next byte the source expect to receive.
- It retransmits the bytes not acknowledged with in specified time period.

TCP Services

TCP offers following services to the processes at the application layer:

- Stream Delivery Service
- Sending and Receiving Buffers
- Bytes and Segments
- Full Duplex Service
- Connection Oriented Service
- Reliable Service

Stream Deliver Service

TCP protocol is stream oriented because it allows the sending process to send data as stream of bytes and the receiving process to obtain data as stream of bytes.

Sending and Receiving Buffers

It may not be possible for sending and receiving process to produce and obtain data at same speed, therefore, TCP needs buffers for storage at sending and receiving ends.

Bytes and Segments

The Transmission Control Protocol (TCP), at transport layer groups the bytes into a packet. This packet is called segment. Before transmission of these packets, these segments are encapsulated into an IP datagram.

Full Duplex Service

Transmitting the data in duplex mode means flow of data in both the directions at the same time.

Connection Oriented Service

TCP offers connection oriented service in the following manner:

1. TCP of process-1 informs TCP of process – 2 and gets its approval.
2. TCP of process – 1 and TCP of process – 2 and exchange data in both the two directions.
3. After completing the data exchange, when buffers on both sides are empty, the two TCP's destroy their buffers.

Reliable Service

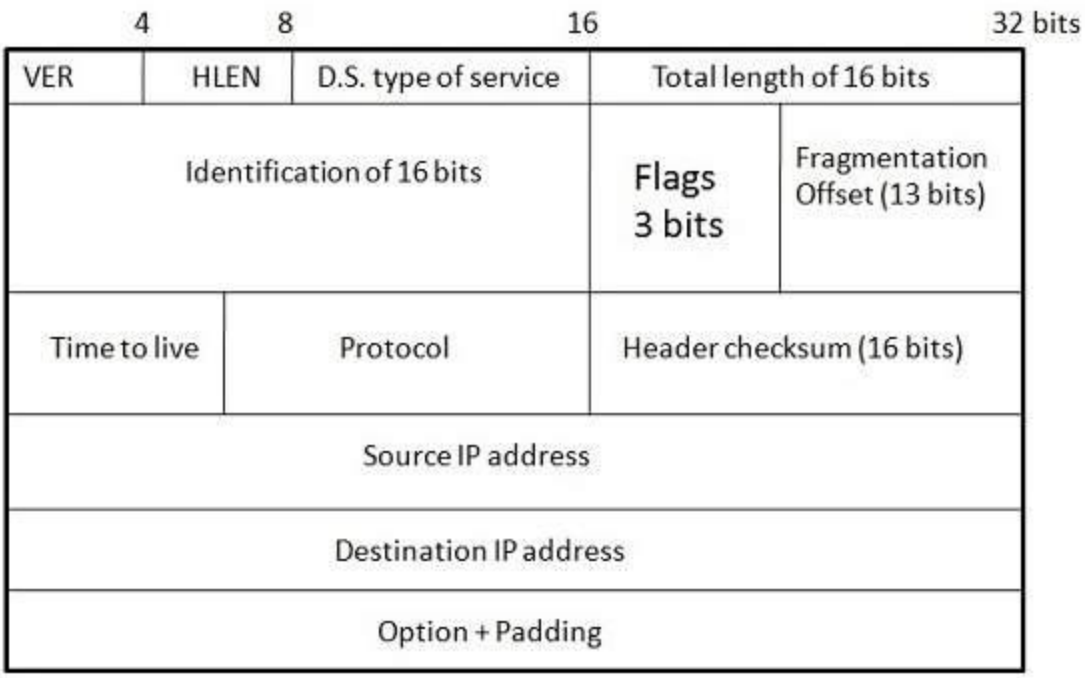
For sake of reliability, TCP uses acknowledgement mechanism.

Internet Protocol (IP)

Internet Protocol is **connectionless** and **unreliable** protocol. It ensures no guarantee of successfully transmission of data.

In order to make it reliable, it must be paired with reliable protocol such as TCP at the transport layer.

Internet protocol transmits the data in form of a datagram as shown in the following diagram:



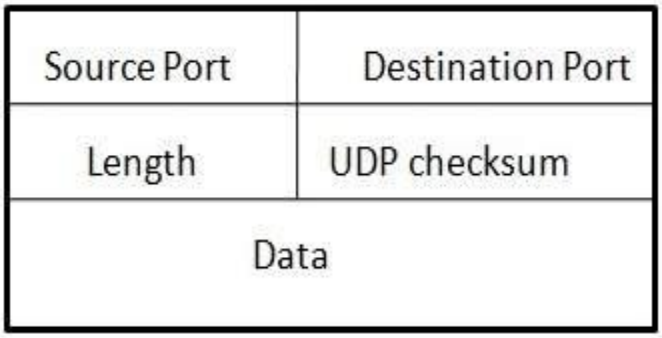
Points to remember:

- The length of datagram is variable.
- The Datagram is divided into two parts: **header** and **data.**
- The length of header is 20 to 60 bytes.
- The header contains information for routing and delivery of the packet.

User Datagram Protocol (UDP)

Like IP, UDP is connectionless and unreliable protocol. It doesn't require making a connection with the host to exchange data. Since UDP is unreliable protocol, there is no mechanism for ensuring that data sent is received.

UDP transmits the data in form of a datagram. The UDP datagram consists of five parts as shown in the following diagram:



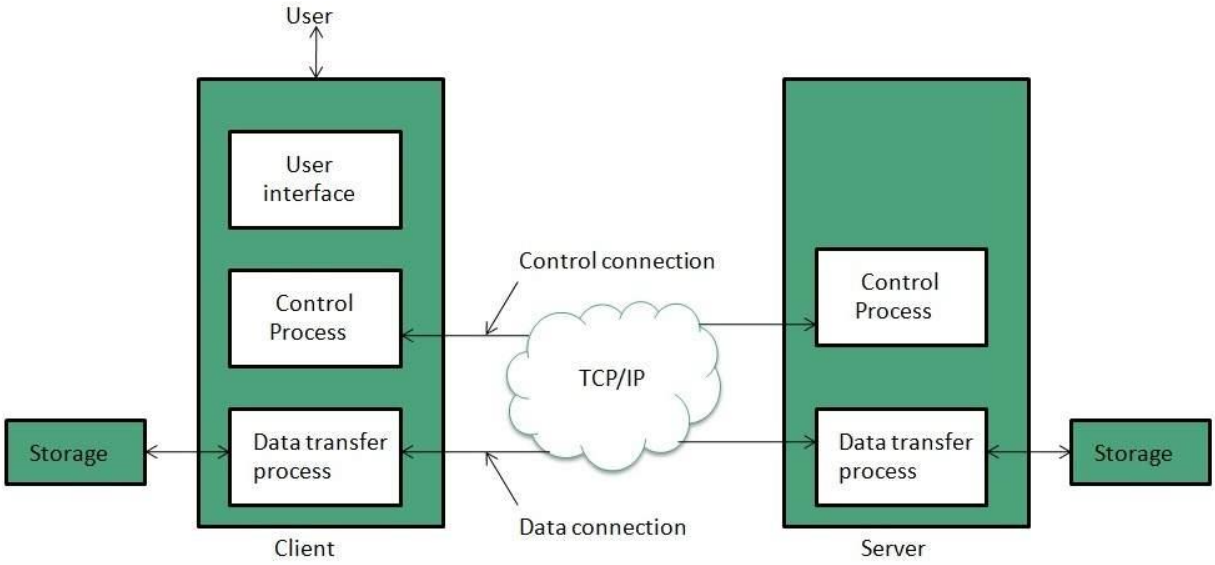
Points to remember:

- UDP is used by the application that typically transmit small amount of data at one time.
- UDP provides protocol port used i.e. UDP message contains both source and destination port number, that makes it possible for UDP software at the destination to deliver the message to correct application program.

File Transfer Protocol (FTP)

FTP is used to copy files from one host to another. FTP offers the mechanism for the same in following manner:

- FTP creates two processes such as Control Process and Data Transfer Process at both ends i.e. at client as well as at server.
- FTP establishes two different connections: one is for data transfer and other is for control information.
- **Control connection** is made between **control processes** while **Data Connection** is made between<="" b="">
- FTP uses **port 21** for the control connection and **Port 20** for the data connection.



Trivial File Transfer Protocol (TFTP)

Trivial File Transfer Protocol is also used to transfer the files but it transfers the files without authentication. Unlike FTP, TFTP does not separate control and data information. Since there is no authentication exists, TFTP lacks in security features therefore it is not recommended to use TFTP.

Key points

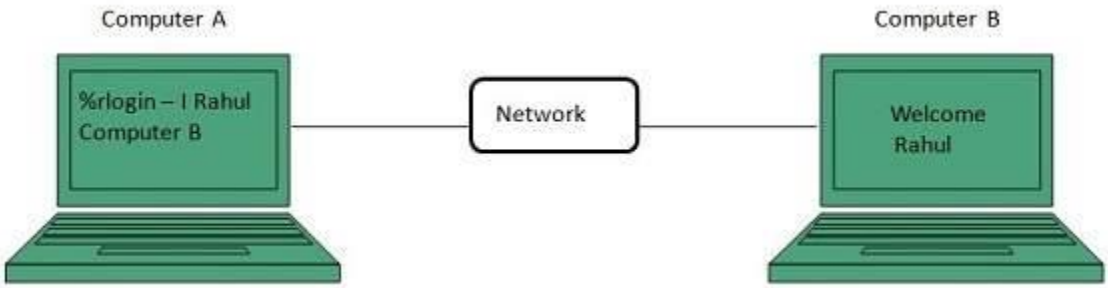
- TFTP makes use of UDP for data transport. Each TFTP message is carried in separate UDP datagram.
- The first two bytes of a TFTP message specify the type of message.
- The TFTP session is initiated when a TFTP client sends a request to upload or download a file.
- The request is sent from an ephemeral UDP port to the **UDP port 69** of an TFTP server.

Difference between FTP and TFTP

S.N.	Parameter	FTP	TFTP
1	Operation	Transferring Files	Transferring Files
2	Authentication	Yes	No
3	Protocol	TCP	UDP
4	Ports	21 – Control, 20 – Data	Port 3214, 69, 4012
5	Control and Data	Separated	Separated
6	Data Transfer	Reliable	Unreliable

Telnet

Telnet is a protocol used to log in to remote computer on the internet. There are a number of Telnet clients having user friendly user interface. The following diagram shows a person is logged in to computer A, and from there, he remote logged into computer B.



Hyper Text Transfer Protocol (HTTP)

HTTP is a communication protocol. It defines mechanism for communication between browser and the web server. It is also called request and response protocol because the communication between browser and server takes place in request and response pairs.

HTTP Request

HTTP request comprises of lines which contains:

- Request line
- Header Fields
- Message body

Key Points

- The first line i.e. the **Request line** specifies the request method i.e. **Get** or **Post.**
- The second line specifies the header which indicates the domain name of the server from where index.htm is retrieved.

HTTP Response

Like HTTP request, HTTP response also has certain structure. HTTP response contains:

- Status line
- Headers
- Message body

The characteristics of interprocess communication:-

The general characteristics of interprocess communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.

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 a *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 *nonblocking* 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

In a system environment such as Java, which supports multiple threads in a single process, the blocking *receive* has no disadvantages, for it can be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. 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. For these reasons, today's systems do not generally provide the nonblocking form of *receive*.

Message destinations • Chapter 3 explains that in the Internet protocols, 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 • As far as the validity property is concerned, 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 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.

Sockets:-

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors. In Unix, every I/O action is done by writing or reading a file descriptor. A file descriptor is just an integer associated with an open file and it can be a network connection, a text file, a terminal, or something else.

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

Sockets were first introduced in 2.1BSD and subsequently refined into their current form with 4.2BSD. The sockets feature is now available with most current UNIX system releases.

Where is Socket Used?

A Unix Socket is used in a client-server application framework. A server is a process that performs some functions on request from a client. Most of the application-level protocols

like FTP, SMTP, and POP3 make use of sockets to establish connection between client and server and then for exchanging data.

Socket Types

There are four types of sockets available to the users. The first two are most commonly used and the last two are rarely used.

Processes are presumed to communicate only between sockets of the same type but there is no restriction that prevents communication between sockets of different types.

- **Stream Sockets** − Delivery in a networked environment is guaranteed. If you send through the stream socket three items "A, B, C", they will arrive in the same order − "A, B, C". These sockets use TCP (Transmission Control Protocol) for data transmission. If delivery is impossible, the sender receives an error indicator. Data records do not have any boundaries.

- **Datagram Sockets** − Delivery in a networked environment is not guaranteed. They're connectionless because you don't need to have an open connection as in Stream Sockets − you build a packet with the destination information and send it out. They use UDP (User Datagram Protocol).

- **Raw Sockets** − These provide users access to the underlying communication protocols, which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more cryptic facilities of an existing protocol.

- **Sequenced Packet Sockets** − They are similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as a part of the Network Systems (NS) socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the Sequence Packet Protocol (SPP) or Internet Datagram Protocol (IDP) headers on a packet or a group of packets, either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets.

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 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 216 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.

Generally, an application, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

Blocking: Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication (a non-blocking *receive* is an option in some implementations). 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 outstanding or future invocation of *receive* on that socket. Messages are discarded at the destination if no process already has a socket bound to the destination port.

***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. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

***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. It is possible to connect a datagram socket 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.

Failure model for UDP datagrams • A failure model for communication channels and defines reliable communication 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. To simplify the discussion, we regard send-omission and receive-omission failures as omission failures in the communication channel.

Ordering: Messages can sometimes be delivered out of sender order. Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

Use of UDP • For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the 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:

Datagram packet

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.

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

```
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();}
   }
}
```

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. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a *SocketException* if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.

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

TCP stream communication and Marshalling:-

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 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. Applications can, if necessary, force data to be sent immediately.

***Lost messages*:** The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), 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. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.

***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 establish 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. This could be a considerable overhead for a single client-server request and reply.

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

ServerSocket: 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. The result of executing *accept* is an instance of *Socket* – a socket to use for communicating with the client.

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. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

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


```
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]);          // UTF is a string encoding see Sn 4.3
                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){System.out.println("close:"+e.getMessage());}}
        }
}
```

```
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]);          // UTF is a string encoding see Sn 4.3
                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){System.out.println("close:"+e.getMessage());}}
        }
}
```

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

```
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());}
   }
}

// this figure continues on the next slide
```

```
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*/}}
            }
}
```

The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences 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; if the two computers are known to be the same type, the conversion to external format can be omitted.

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. Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. 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 to external data representation and marshalling are discussed:

CORBA's 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. It can be used by a variety of programming languages.

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 fomat for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part 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. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures.

In the first two approaches, the primitive data types are marshalled 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. The HTTP protocol, which is described in Chapter 5, is another example of the textual approach.

Another issue with regard to the design of marshalling methods is whether the marshalled 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*.

Although we are interested in the use of an external data representation for the arguments and results of RMIs and RPCs, it does have a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.

Client-Server Communication In Distributed Systems:-

Today we will get familiar with client-server paradigm and common communication methods used for it: ***sockets***, ***RPC*** **and** ***pipes***.

Client-server

Client-server concept underpins distributed systems over a couple of decades. There are two counterparts in the concept: a *client* and a *server*. In practice there are often multiple clients and single server. Clients start communication by sending requests to the server, the server handles them and usually returns responses back.

Client processes often do not live long, while server process, which is sometimes called *daemon*, live till OS shutdown.

Sockets

IPC with sockets is very common in distributed systems. In nutshell, a socket is a pair of an IP address and a port number. For two process to communicate, each of them needs a socket.

When server daemon is running on a host, it is listening to its port and handles all requests sent by clients to the port on the host (server socket). A client must know IP and port of the server (server socket) to send a request to it. Client's port is often provided by OS kernel when client starts communication with the server and is freed when communication is over.

Although communication using sockets is common and efficient, it is considered low level, because sockets only allow to transfer unstructured stream of bytes between processes. It is up to client and server applications to impose a structure on the data passed as byte stream.

Remote Procedure Calls

RPC is a higher level communication method. It was designed to mimic procedure call mechanism, but execute it over network. RPC is conceptually similar to message passing IPC and is usually built on top of socket communication.

In contrast to IPC messages, RPC messages are well structured. Each message includes information about function to be executed and the parameters to be passed to that function. When the function is executed, a response with output is sent back to the requester in a separate message.

RPC hides the details of communication by providing a stub on the client side. When client needs to invoke a remote procedure, it invokes the stub and pass it the parameters. The stub marshals the parameters and sends a message to RPC daemon running on the server. RPC daemon (a similar stub on the server side) receives the message and invokes the procedure on the server. Return values are passed back to the client using the same technique.

Pipes

Pipe is one of the oldest and simplest IPC methods, that appeared in early UNIX systems. A pipe is an IPC abstraction with two endpoints, similar to a physical pipe. Usually one process puts data to one end of the pipe and another process consumes them from the other one.

Ordinary pipes

Ordinary pipes are unidirectional, i.e. allow only one-way communication. They implement standard producer-consumer mechanism, where one process writes to the pipe and another one reads from it. For two-way communication two pipes are needed.

Ordinary pipes require a parent–child relationship between communicating processes, because a pipe can only be accessed from process that created or inherited it. Parent process creates a pipe and uses it to communicate with a child created via *fork()*. Once communication is over and processes terminated, the ordinary pipe ceases to exist.

Named pipes

Named pipes are more powerful. They do not require parent–child relationship and can be bidirectional. Once a named pipe is created, multiple non related processes can communicate over it.

Named pipe continues to exist after communicating processes have terminated. It must be explicitly deleted when not required anymore.

Named pipes are known as FIFOs in UNIX systems and appear as files in the file system.

Group communication

A multicast operation is more appropriate – this 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.

Group communication:-

A *multicast operation* is more appropriate – this 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. There is a range of possibilities in the desired behaviour of a multicast. The simplest multicast rotocol provides no guarantees about message delivery or ordering.

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

***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.

***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.

***Better performance through replicated data*:** Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value is multicast to the processes managing the replicas.

***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. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers.

Multicast and Group Communication:-

Overview

A common operation in many applications is to deliver the same message to multiple recipients. Your booksite CDN (content distribution network) is a good example of this. When a new book is added to the site it needs to be distributed to all of the mirror sites. One way of doing this is to have the main server send the new content to each of the mirrors -- multiple unicast. This approach puts a significant burden on the server. IP multicast and overlay multicast provide mechanisms for delivering the same message to multiple recipients in a more efficient manner.

More generally, group communication refers to the idea of enabling a process to communicate with a group of processes, typically without knowing which processes belong to the group. Group communication is often concerned with reliable delivery of messages and the order in which messages are received at each process.

IP Multicast:-

IP Multicast enables one-to-many communication at the network layer. A source sends a packet destined for a *group* of other hosts and the intermediate routers take care of replicating the packet as necessary. The intermediate routers are also responsible for determining which hosts are members of the group. IP Multicast uses UDP for communication, therefore it is unreliable.

To join a multicast group, a host sends a join message, using the Internet Group Management Protocol (IGMP), to its first-hop router. Groups are identified by a single class D IP address

(in the range 224.0.0.0 to 239.255.255.255). In this way, messages destined for a group are addressed to the appropriate IP address, just like any other message.

In order to deliver a single message to several destinations, the routers that connect the members of the group organize into a tree. There are several algorithms for determining the edges of the tree. A *shared tree* is formed by selecting a center node. When a router receives a join from an attached host, it sends a join message toward the center node for the given group. Each router that receives the join will note that a downstream router is joined to the given group. The join will stop when it reaches the center, or if it reaches another router that is already part of the shared tree.

Overlay Multicast

Unfortunately, IP multicast is not widely supported. Though not as efficient, overlay multicast achieves many of the same benefits without requiring support from ISPs. In an overlay multicast scheme, end hosts organize into a delivery tree. Each host receives content from its parent and delivers content to its children. In this way, the load of distributing content is shared by all of the members of the group.

The main components of an overlay multicast application are the **join** algorithm and the procedure for ensuring **fault tolerance**.

Join Algorithm

Joining an overlay multicast group involves determining an appropriate parent and possibly a set of children. This is not unlike joining a peer-to-peer network, which we'll discuss in a few weeks. One method, derived from the Host Multicast Tree Protocol (HMTP), is to use a rendezvous point (RP). A new node (*n*) queries the RP for the root of the tree. *n* begins at the root and recursively walks down the tree searching for an appropriate place to join. At each iteration, *n* determines whether the current node is an appropriate parent, for example by asking whether it can accept new children and/or measuring the RTT to reach it. *n* either selects the current node as a parent, or asks it for its set of children and repeats the process for each child.

Fault Tolerance

Nodes may fail at any time, therefore, detecting and tolerating faults is extremely important. Typically, nodes exchange "keep alive" messages to notify neighbors that they are still alive. In HMTP, child nodes periodically send a REFRESH message to parents. If a parent fails to receive a REFRESH message in a given period of time, it can remove the given child from its child list. If a child attempts to contact a parent and cannot, it must find a new parent. One option is for the parentless child to reinitiate the join procedure. For efficiency, a child can also cache information about nodes it tried during its initial join procedure and attempt to join to nodes it previously identified as potential good parents.

Group Communication and Reliable Multicast:-

Multicast is essentially an implementation of group communication.

The basic model for multicast supports the following operations:

To B-multicast (g, m): for each process p in g, send(p, m);

On receive(m) at p: B-deliver(m) at p.

If the communication channels between processes are unreliable, it may be desirable to add reliability to basic multicast. Reliability typically involves some kind of acknowledgement scheme where processes acknowledge receipt of a particular message. In practice, this can result in ACK implosion; a process sends 1 message and receives n-1 responses where n is the total number of processes in the group. To reduce the number of acknowledgements required, a negative acknowledgement scheme can be used.

Basic, reliable multicast does not ensure that messages are delivered to the application in any particular order. If a message is delayed in the network, it may arrive after another message that was actually sent first. Depending on the application, there are several possible ordering requirements a developer might enforce:

FIFO ordering: If a correct process issues multicast(g, m) and then multicast(g, m'), then every correct process that delivers m' will deliver m before m'.

Causal ordering: If multicast(g, m) → multicast(g, m'), where → is the happened-before relation induced only by messages sent between the members of g, then any correct process that delivers m' will deliver m before m'.

Total ordering: If a correct process delivers message m before it delivers m', then any other correct process that delivers m' will deliver m before m'.

FIFO ordering can be achieved by having each process maintain timestamps for the messages that it sends to the group. Total ordering can be achieved by using a separate sequencer process that assigns sequence numbers or by using a distributed algorithm whereby a process sends a message, the remaining processes propose a sequence number for the message, and the sender selects the highest sequence number proposed. Causal ordering can be achieved by using vector timestamps.