Unit-2 Interprocess Communication

Syllabus: 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.

Topic 01: INTRODUCTION

- The java API for interprocess communication in the internet provides both datagram and stream communication.
- The two communication patterns that are most commonly used in distributed programs:
  - Client-Server communication
    - The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).
  - Group communication
    - The same message is sent to several processes.

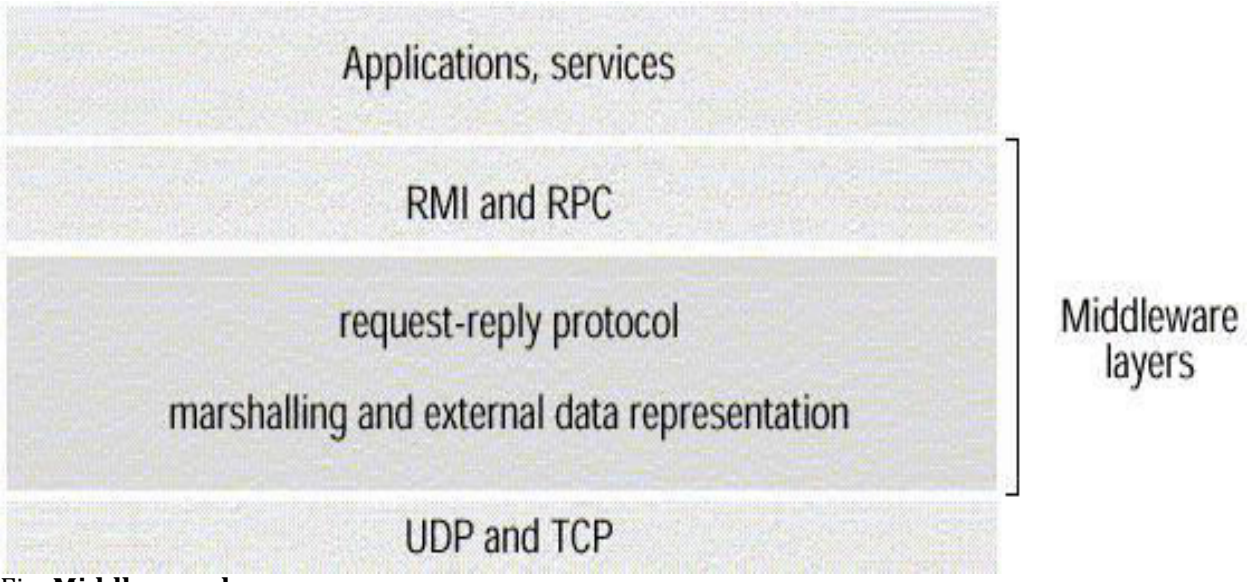

Fig: **Middleware layers**

- Remote Method Invocation (RMI)
  - It allows an object to invoke a method in an object in a remote process.
    - E.g. CORBA and Java RMI
- Remote Procedure Call (RPC)
  - It allows a client to call a procedure in a remote server.
- The application program interface

(API) to UDP provides a message passing abstraction.

- Message passing is the simplest form of interprocess communication.
- API enables a sending process to transmit a single message to a receiving process.
- The independent packets containing theses messages are called datagrams.
- In the Java and UNIX APIs, the sender specifies the destination using a socket.
- Socket is an indirect reference to a particular port used by the destination process at a destination computer.

- The application program interface (API) 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.
- Request-reply protocols are designed to support client-server communication in the form of either RMI or RPC.
- Group multicast protocols are designed to support group communication.
- Group multicast is a form of interprocess communication in which one process in a group of processes transmits the same message to all members of the group.

Topic No : 02 : The API for the Internet Protocols

Sub Topic 2.1 : The Characteristics of Interprocess Communication

- Synchronous and asynchronous communication
  - In the synchronous form, both send and receive are blocking operations.
  - In the asynchronous form, the use of the send operation is non-blocking and the receive operation can have blocking and non-blocking variants.
- Message destinations
  - A local port is a message destination within a computer, specified as an integer.
  - A port has an exactly one receiver but can have many senders.
- Reliability
  - A reliable communication is defined in terms of validity and integrity.
  - A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost.
  - For integrity, messages must arrive uncorrupted and without duplication.

- Ordering
  - Some applications require that messages be delivered in sender order.

Sub Topic 2.2 : Sockets

- Internet IPC mechanism of Unix and other operating systems (BSD Unix, Solaris, Linux, Windows NT, Macintosh OS)
- Processes in the above OS can send and receive messages via a socket.
- Sockets need to be bound to a port number and an internet address in order to send and receive messages.
- Each socket has a transport protocol (TCP or UDP).
- Messages sent to some internet address and port number can only be received by a process using a socket that is bound to this address and port number.
- Processes cannot share ports (exception: TCP multicast).
- Both forms of communication, UDP and TCP, use the socket abstraction, which provides and endpoint for communication between processes.
- Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process.

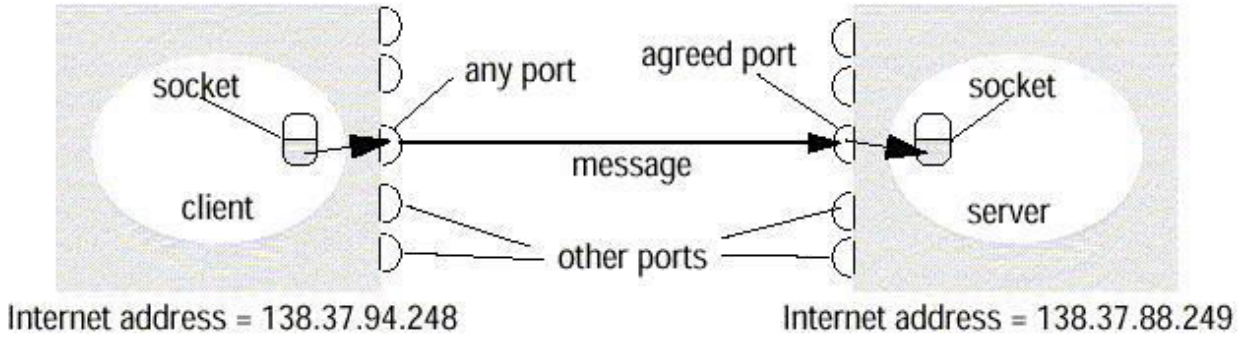

Figure . Sockets and ports

Sub Topic 2.3: UDP Datagram Communication

- UDP datagram properties
  - No guarantee of order preservation
  - Message loss and duplications are possible
- Necessary steps
  - Creating a socket
  - Binding a socket to a port and local Internet address

- ❖ A client binds to any free local port
- ❖ A server binds to a server port

- Receive method
  - ➢ It returns Internet address and port of sender, plus message.
- Issues related to datagram communications are:
  - ➢ Message size
    - ❖ IP allows for messages of up to 216 bytes.
    - ❖ Most implementations restrict this to around 8 kbytes.
    - ❖ Any application requiring messages larger than the maximum must fragment.
    - ❖ If arriving message is too big for array allocated to receive message content, truncation occurs.
  - ➢ Blocking
    - ❖ Send: non-blocking
      - upon arrival, message is placed in a queue for the socket that is bound to the destination port.
    - ❖ Receive: blocking
      - Pre-emption by timeout possible
      - If process wishes to continue while waiting for packet, use separate thread
  - ➢ Timeout
  - ➢ Receive from any
- UDP datagrams suffer from following failures:
  - ➢ Omission failure
  - ➢ Messages may be dropped occasionally,
  - ➢ Ordering

Java API for UDP Datagrams

- The Java API provides datagram communication by two classes:
  - ➢ DatagramPacket
    - ❖ It provides a constructor to make an array of bytes comprising:

- Message content
- Length of message
- Internet address
- Local port number

❖ It provides another similar constructor for receiving a message.

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

➢ DatagramSocket

❖ This class supports sockets for sending and receiving UDP datagram.

❖ It provides a constructor with port number as argument.

❖ No-argument constructor is used to choose a free local port.

❖ DatagramSocket methods are:

- send and receive
- setSoTimeout
- connect

➢ Example

➢ – The process creates a socket, sends a message to a server at port 6789 and waits to receive a reply.

```
import java.net.*;
import java.io.*;
public class UDPClient{
public static void main(String args[]){
// args give message contents and destination hostname
try {
DatagramSocket aSocket = new DatagramSocket(); // create socket
byte [] m = args[0].getBytes();
InetAddress aHost = InetAddress.getByName(args[1]); // DNS lookup
int serverPort = 6789;
DatagramPacket request =
```

```
new DatagramPacket(m, args[0].length(), aHost, serverPort);

aSocket.send(request); //send nessage byte[] buffer = new

byte[1000];

DatagramPacket reply = new DatagramPacket(buffer, buffer.length);

aSocket.receive(reply); //wait for reply System.out.println("Reply: "

+ new String(reply.getData())); aSocket.close();


}catch (SocketException e){System.out.println("Socket: " + e.getMessage());

}catch (IOException e){System.out.println("IO: " + e.getMessage());}

} finally{if (aSocket !=null)aSocket.close()}

}

}
```

Figure. UDP client sends a message to the server and gets a reply (Above Java Code)

Example

– The process creates a socket, bound to its server port 6789 and waits to receive a request message from a 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()} }

}
```

Figure. UDP server repeatedly receives a request and sends it back to the client (Above Java Code)

Sub Topic 2.4: TCP Stream Communication

- The API to the TCP protocol provides the abstraction of a stream of bytes to be written to or read from.
  - Characteristics of the stream abstraction:
    - Message sizes
    - Lost messages
    - Flow control
    - Message destinations
- Use of TCP
  - Many services that run over TCP connections, with reserved port number are:
    - HTTP (Hypertext Transfer Protocol)
    - FTP (File Transfer Protocol)
    - Telnet
    - SMTP (Simple Mail Transfer Protocol)
  - Matching of data items
  - Blocking
  - Threads
- Java API for TCP streams

- The Java interface to TCP streams is provided in the classes:
  - ServerSocket
    - It is used by a server to create a socket at server port to listen for connect requests from clients.
  - Socket
    - It is used by a pair of processes with a connection.
    - The client uses a constructor to create a socket and connect it to the remote host and port of a server.
    - It provides methods for accessing input and output streams associated with a socket.

- Example
- – The client process creates a socket, bound to the hostname and server port 6789.

NO CODE….FIGURE

Example

– The server process opens a server socket to its server port 6789 and listens for connect requests.

```
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 socket:"+e.getMessage());}
}
}
```

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

```
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("readline:"+e.getMessage());}
} finally {try{clientSocket.close();}catch(IOException e){/*close failed*/}}
}
}
```

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

Topic No 3: **External Data Representation**

- The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes.
- Irrespective of the form of communication used, the data structure must be converted to a sequence of bytes before transmission and rebuilt on arrival.

- External Data Representation is an agreed standard for the representation of data structures and primitive values.
- Data representation problems are:
  - Using agreed external representation, two conversions necessary
  - Using sender's or receiver's format and convert at the other end
- Marshalling
  - Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- Unmarshalling
  - Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.
- Three approaches to external data representation and marshalling are:
  - CORBA
  - Java's object serialization
  - XML
- Marshalling and unmarshalling activities is usually performed automatically by middleware layer.
- Marshalling is likely error-prone if carried out by hand.

Sub Topic 3.1: CORBA Common Data Representation (CDR)

- CORBA Common Data Representation (CDR)
  - CORBA CDR is the external data representation defined with CORBA 2.0.
  - It consists 15 primitive types:
    - Short (16 bit)
    - Long (32 bit)
    - Unsigned short
    - Unsigned long
    - Float(32 bit)
    - Double(64 bit)
    - Char
    - Boolean(TRUE,FALSE)

- Octet(8 bit)
- Any(can represent any basic or constructed type)

- Composite type are shown in Figure

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can 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

Figure. CORBA CDR for constructed types

- Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in above table / Figure.
- Below Figure/ table shows a message in CORBA CDR that contains the three fields of a struct whose respective types are string, string, and unsigned long.

- example: struct with value {'Smith', 'London', 1934}

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	1934	*unsigned long*

Figure. CORBA CDR message

Sub topic 3.2 :Java object serialization

- In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.
- An object is an instance of a Java class.
    - Example, the Java class equivalent to the Person struct

```
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
}
```

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
1934	5 Smith	6 London	h1	values of instance variables

Figure '. Indication of Java serialization form

Remote Object References

- Remote object references are needed when a client invokes an object that is located on a remote server.
- A remote object reference is passed in the invocation message to specify which object is to be invoked.

- Remote object references must be unique over space and time.
- In general, may be many processes hosting remote objects, so remote object referencing must be unique among all of the processes in the various computers in a distributed system.
- generic format for remote object references is shown in below Figure.

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Figure : Representation of a remote object references

- internet address/port number: process which created object
- time: creation time
- object number: local counter, incremented each time an object is created in the creating process
- interface: how to access the remote object (if object reference is passed from one client to another)

Topic No 4: **Client-Server Communication**

- The client-server communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.
- Often built over UDP datagrams
- Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary
- Avoidance of connection establishment overhead
- No need for flow control due to small amounts of data are transferred
- The request-reply protocol was based on a trio of communication primitives: doOperation, getRequest, and sendReply shown in Figure .

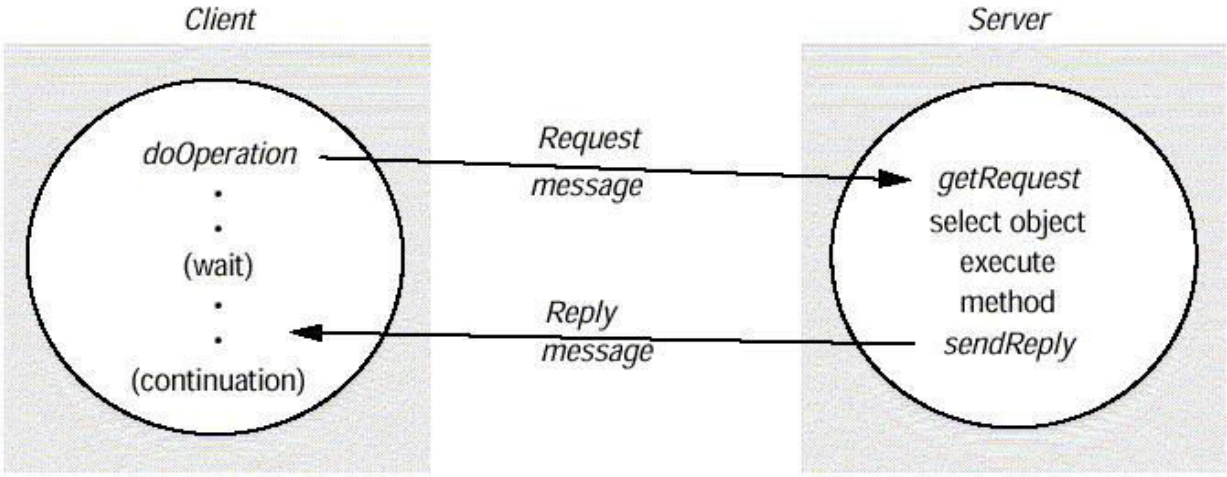

Figure:. Request-reply communication

- The designed request-reply protocol matches requests to replies.
- If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.
- Figure outlines the three communication primitives.

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

public byte[] getRequest ();
acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
sends the reply message *reply* to the client at its Internet address and port.

Figure :. Operations of the request-reply protocol

- The information to be transmitted in a request message or a reply message is shown in Figure .

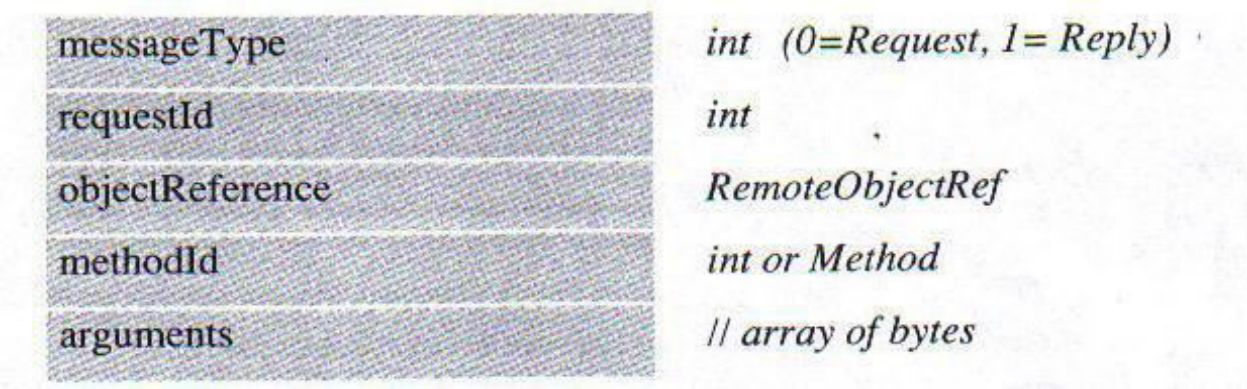

Figure : Request-reply message structure

- In a protocol message
  - The first field indicates whether the message is a request or a reply message.
  - The second field request id contains a message identifier.
  - The third field is a remote object reference .
  - The forth field is an identifier for the method to be invoked.
- Message identifier
  - A message identifier consists of two parts:
    - A requestId, which is taken from an increasing sequence of integers by the sending process
    - An identifier for the sender process, for example its port and Internet address.
- Failure model of the request-reply protocol
  - If the three primitive doOperation, getRequest, and sendReply are implemented over UDP datagram, they have the same communication failures.
    - Omission failure
    - Messages are not guaranteed to be delivered in sender order.
- RPC exchange protocols
  - Three protocols are used for implementing various types of RPC.
    - The request (R) protocol.
    - The request-reply (RR) protocol.
    - The request-reply-acknowledge (RRA) protocol.

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

Figure. RPC exchange protocols

- In the R protocol, a single request message is sent by the client to the server.
- The R protocol may be used when there is no value to be returned from the remote method.
- The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol.
- RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply.
- HTTP: an example of a request-reply protocol
  - HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.
  - HTTP protocol steps are:
    - Connection establishment between client and server at the default server port or at a port specified in the URL
    - client sends a request
    - server sends a reply
    - connection closure
  - HTTP 1.1 uses persistent connections.
    - Persistent connections are connections that remains open over a series of request-reply exchanges between client and server.
  - Resources can have MIME-like structures in arguments and results.
  - A Mime type specifies a type and a subtype, for example:

- text/plain
- text/html
- image/gif
- image/jpeg

- HTTP methods
  - GET
    - Requests the resource, identified by URL as argument.
    - If the URL refers to data, then the web server replies by returning the data
    - If the URL refers to a program, then the web server runs the program and returns the output to the client.

method	*URL*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure :. HTTP request message

  - HEAD
    - This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)
  - POST
    - Specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request.
    - This method is designed to deal with:
      - Providing a block of data to a data-handling process
      - Posting a message to a bulletin board, mailing list or news group.
      - Extending a dataset with an append operation
  - PUT
    - Supplied data to be stored in the given URL as its identifier.
  - DELETE
    - The server deletes an identified resource by the given URL on the server.
  - OPTIONS

- ❖ A server supplies the client with a list of methods.
- ❖ It allows to be applied to the given URL

➢ TRACE

- ❖ The server sends back the request message

➢ A reply message specifies

- ❖ The protocol version
- ❖ A status code
- ❖ Reason
- ❖ Some headers
- ❖ An optional message body

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Topic No 5: **Group Communication**

- The pairwise exchange of messages is not the best model for communication from one process to a group of other processes.
- A multicast operation is more appropriate.
- Multicast operation is an operation that sends a single message from one process to each of the members of a group of processes.
- The simplest way of multicasting, provides no guarantees about message delivery or ordering.
- Multicasting has 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.
  - Finding the discovery servers 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.
- Propagation of event notifications
    - Multicast to a group may be used to notify processes when something happens.

Sub Topic 5.1: 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 multicast group.
- A multicast group is specified by class D IP address for which first 4 bits are 1110 in IPv4.
- The membership of a multicast group is dynamic.
- A computer belongs to a multicast group if one or more processes have sockets that belong to the multicast group.
- The following details are specific to IPv4:
    - Multicast IP routers
        - IP packets can be multicast both on local network and on the wider Internet.
        - Local multicast uses local network such as Ethernet.
        - 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
        - Multicast addressing may be permanent or temporary.
        - Permanent groups exist even when there are no members.
        - Multicast addressing by temporary groups must be created before use and cease to exit when all members have left.
        - The session directory (sd) program can be used to start or join a multicast session.
        - session directory provides a tool with an interactive interface that allows users to browse advertised multicast sessions and to advertise their own session, specifying the time and duration.

➢ Java API to IP multicast

- The Java API provides a datagram interface to IP multicast through the class MulticastSocket, which is a subset of DatagramSocket with the additional capability of being able to join multicast groups.
- The class MulticastSocket provides two alternative constructors , allowing socket to be creative to use either a specified local port, or any free local port.

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
   public static void main(String args[]){
    // args give message contents and 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();}
 }

}
```

Figure :. Multicast peer joins a group and sends and receives datagrams

- A process can join a multicast group with a given multicast address by invoking the joinGroup method of its multicast socket.
- A process can leave a specified group by invoking the leaveGroup method of its multicast socket.
- The Java API allows the TTL to be set for a multicast socket by means of the setTimeToLive method. The default is 1, allowing the multicast to propagate only on the local network.