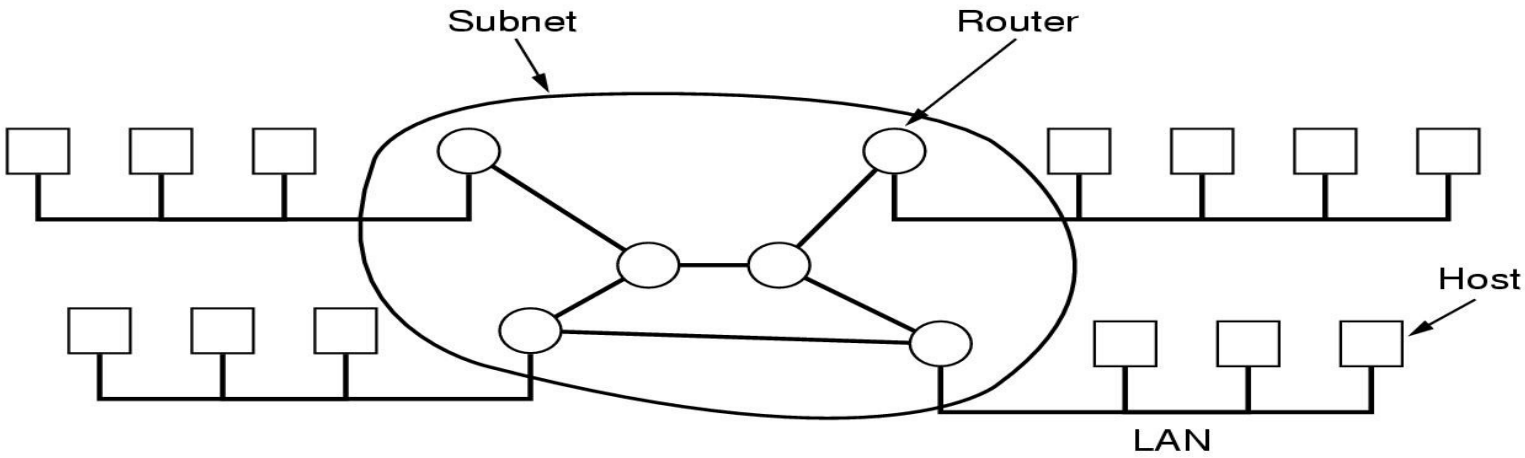
Distributed Systems

Interprocess Communication

Network Architecture

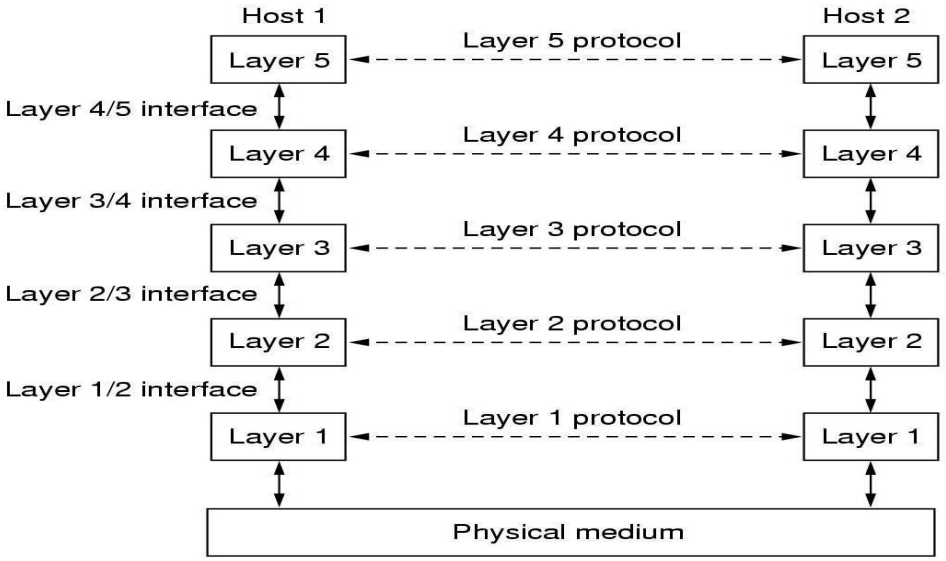
- Network Architecture: framework for designing and implementing Networks
- Components:
  - Software
    - Protocols
    - Services
  - Hardware
    - Transmission technology, media and devices
    - Scale: LANs, MANs, and WANs
    - Topology

Subnet



Layered Architecture

- The purpose of each layer is to offer a communication services to higher level layers
- Each layer has two interfaces
  - peer-to-peer interface
    - defines the form and types of messages exchanged between peers (indirect communication)
  - service interface
    - defines the primitives (operations) that a layer provides to the layer above it
- Layering is non-linear



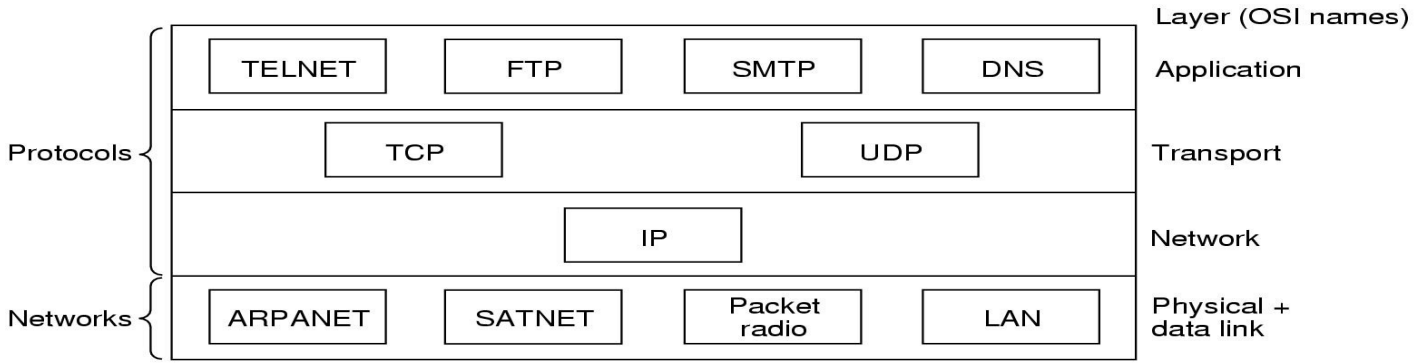
Protocols

- The functionality encapsulated within each layer is called *Protocol*
- The Protocol refers to both
  - the abstract peer-to-peer & service interfaces and
  - the objects that implement those interfaces
- Protocol vs. Service
  - Service is the set of primitives provided to the higher layer
  - Protocol defines the implementation of these primitives
- Protocol stack

ISO OSI Architecture

- International Standards Organisation (ISO)
- *Physical*: transmission of raw bits onto the communications medium
- *Data link*: reliable transmission of frames, flow control, arbitration
- *Network*: packet switching, routing congestion control
- *Transport*: process-to-process channel, node-to-node connection, provides user services, flow control, multiplexing
- *Session*
- *Presentation*
- *Application*

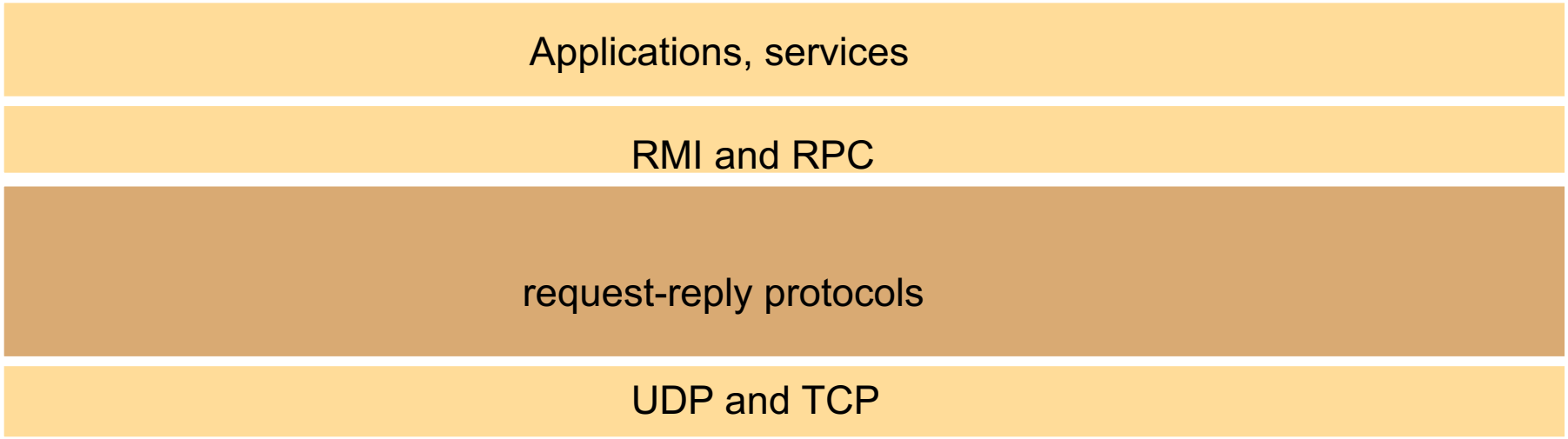
Internet Architecture (TCP/IP)



- Host-to-Network Layer (OSI Physical and Data link layers)
- Internet Layer (OSI Network layer - Internet Protocol/IP)
- **Transport Layer**
  - **Transmission Control Protocol - TCP**
  - **User Datagram Protocol - UDP)**
- Application Layer

Accessing Transport Layer

- How application programs use protocol software (invoke the services) of transport layer to communicate across networks

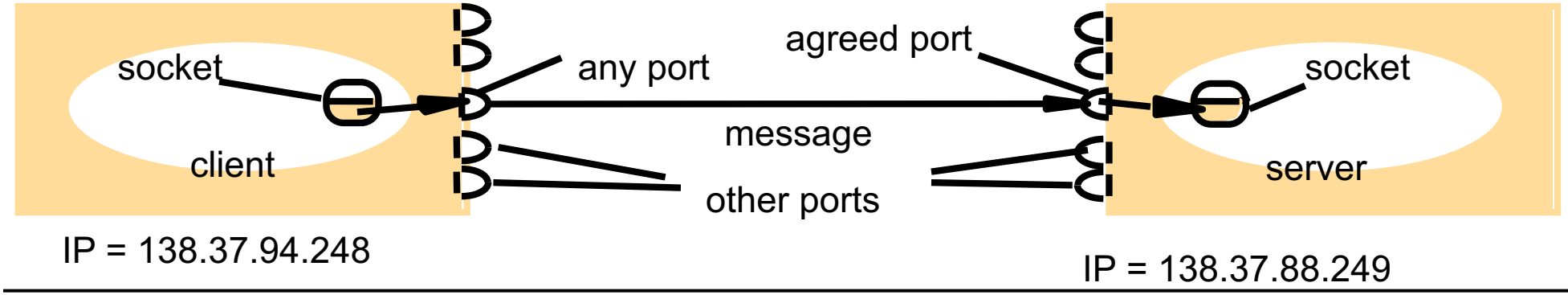


Inter-process Communication

- Message passing model: send - receive
- Synchronous communications : blocking
- Asynchronous communications
  - Blocking (receive) vs non blocking (send, receive)
- Reliability
  - Validity: messages are guaranteed to be delivered despite of packets being lost
  - Integrity: messages must arrive uncorrupted and without duplication
- Ordering: message send and delivery

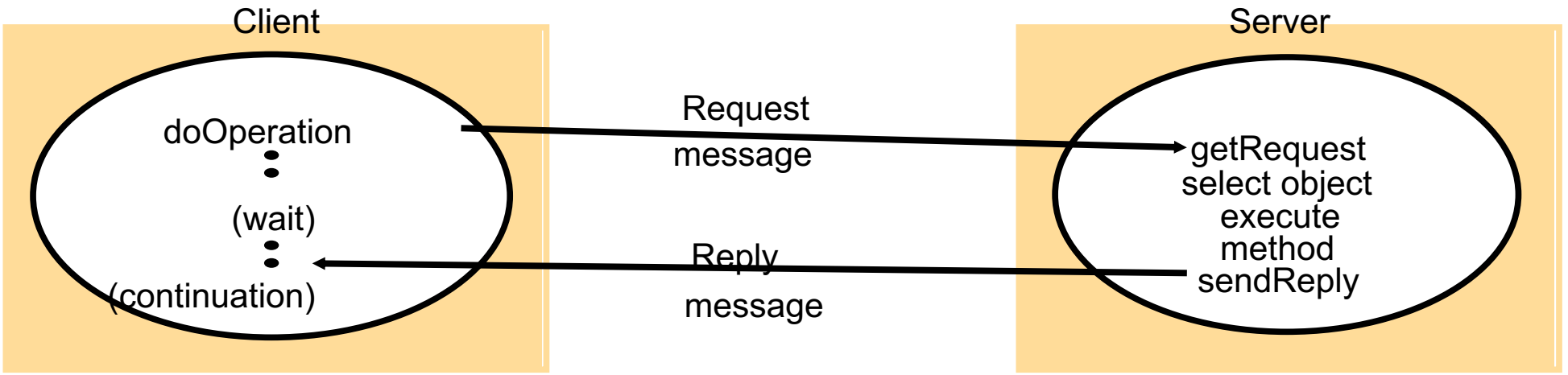
Sockets and ports

- Sockets provide an abstract endpoint for inter-process communication (UDP, TCP)
- Destination:
  - internet address, port, process id
- Port: message destination within a computer (integer) –Internet addresses: names rather than numbers (2^{16})
- A socket must be bound to a local port and an internet address
- A process may use multiple ports but cannot share ports (exception: multicast)



The Request-Reply Protocol

- Processes communicate using a protocol



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.

Issues

- Message identifiers (sequence+sender id)
- Failure Model
- Timeouts
- Duplicate request messages
- Lost reply messages
- History for retransmissions

Java API for Internet addresses

- Class *InetAddress*
  - uses DNS (Domain Name System)

InetAddress aComputer =
InetAddress.getByName("gromit.cs.bham.ac.uk");

  - throws *UnknownHostException*
  - encapsulates detail of IP address (4 bytes for IPv4 and 16 bytes for IPv6)

UDP Sockets

- Receive method returns the IP address+port of sender so that a reply can be sent
- Message size: receiving process specifies a byte-array in which to receive the message (IP: 2^{16} bytes, usually 8kB)
- Non-blocking send & blocking receive (timeouts and threads for deadlocks)
- Receive *from any*

UDP Sockets – Java API

- *DatagramPacket* class provides
  - a constructor that makes an instance of a message: [message byte array, length,IP address,post no]
  - A constructor for use when receiving a message
  - Methods: *getData, getPort, getAddress*
- *DatagramSocket* class provides
  - A constructor that takes a port as an argument for use by processes that need to use a port
  - A no-argument constructor that allows the system to choose a free local port
  - *SocketException* if problem with port
  - Methods:
    - *send:* argument an instance of DatagramPacket
    - *Receive:* argument an empty DatagramPacket
    - *setSoTimeout: InterruptedIOExpception* when expire
    - *connect:* connect a socket to a particular remote port and IP address, the socket is abe to send to and receive from that address

Java API for Datagrams

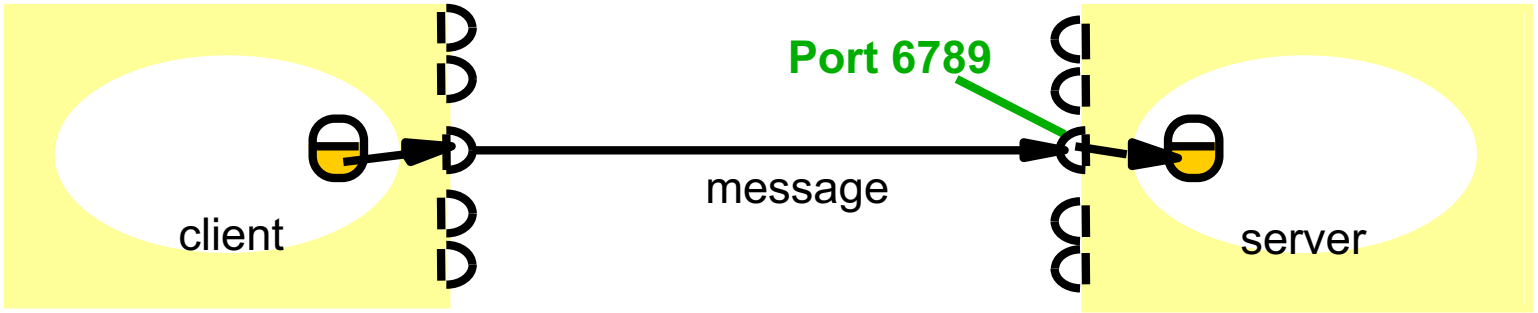
- Simple send/receive, with messages possibly lost/out of order
- Class *DatagramPacket*

message (=array of bytes)	message length	Internet addr	port no

  - packets may be transmitted between sockets
  - packets truncated if too long
  - provides *getData*, *getPort*, *getAddress*

example...

- UDP Client
  - sends a message and gets a reply
- UDP Server
  - repeatedly receives a request and sends it back to the client



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

```
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,  args[0].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 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();}
  }
}
```

TCP Sockets

- API provides an abstraction of a stream of bytes to which data may be written and from which data may be read. The abstraction hides the following:
  - Message sizes. The application can choose how much data data it writes to a stream. TCP may fragment it to smaller packets
  - Lost messages
  - Flow control
  - Message duplication and ordering
  - Message destinations: once a connection is established, processes simply read from and write to a stream (no need for IP addresses and ports)
- Client-server model during connection:
  - client creates a stream socket bound to a port and asks for a connection to a server port
  - server creates a listening socket bound a a port and waits to *accept* connect requests
- During operation each socket is both for input and output
- Close a socket when no more data to write

TCP Sockets – Java API

- *ServerSocket* class: create a socket at a server port for listening for connect requests
  - Method *accept*: wait until there is a *connect* request in the queue, then create an instance of *Socket*
- *Socket* class: client uses a constructor which creates a socket (specifying the DNS hostname and port of a server) and connects it to the specified remote server (*UknownHostException*, *IOException*)
  - Methods: *getInputStream* and *getOutputStream*
    - *R*eturn types are abstract classes that define methods for reading (*InputStream)* and writing (*OutputStream)* bytes
    - Return values can be used as the arguments of constructors for suitable input and output streams.

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());}}
        }
}
```

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());}
    }
}
```

continued

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

Group Communication:

- Multicast: an operation that sends a single message from one process to each of the members of a group of processes
- Fault tolerance based on replicated services
- Finding the discovery servers in spontaneous networking
- Better performance through replicated data
- Propagation of event notifications

IP multicast

- multicast group is specified by a class D Internet address
  - membership is dynamic, to join make a socket
  - programs using multicast use UDP and send datagrams to mutlicast addresses and (ordinary) port
  - (For example of Java code see book)