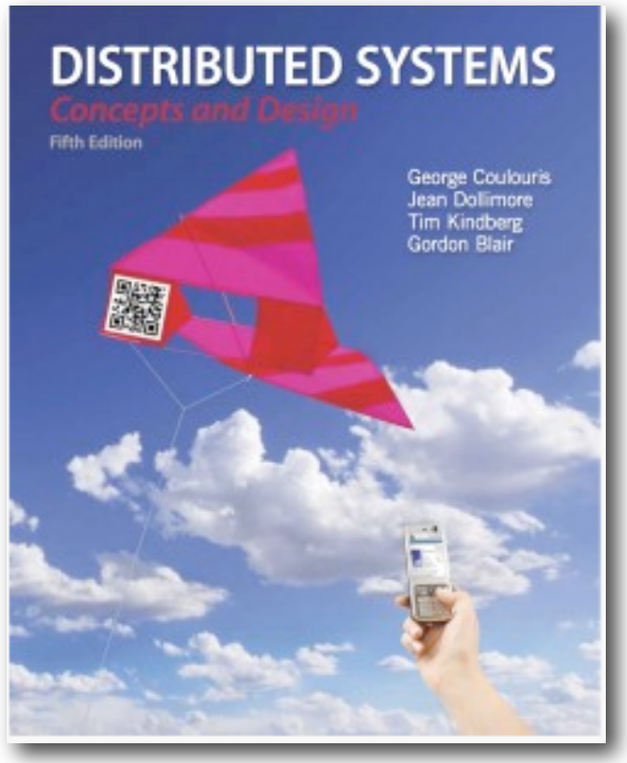
DTU Informatics
Department of Informatics and Mathematical Modelling

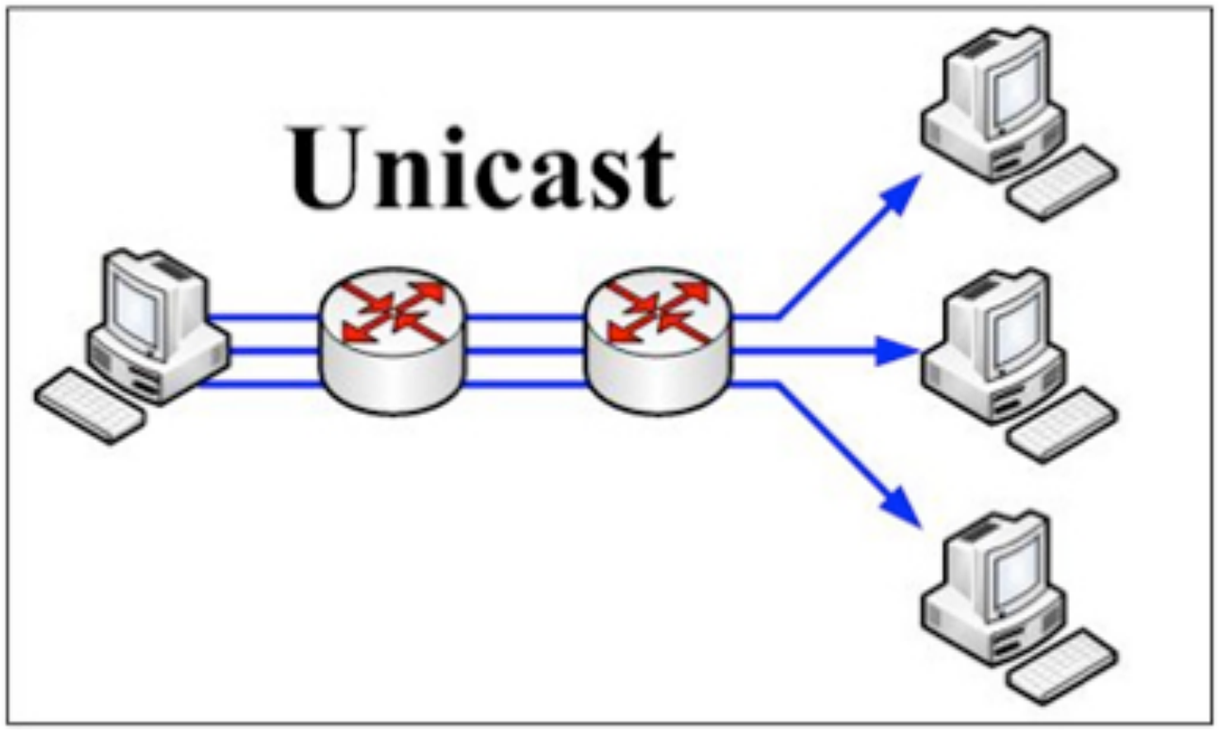
Interprocess Communication

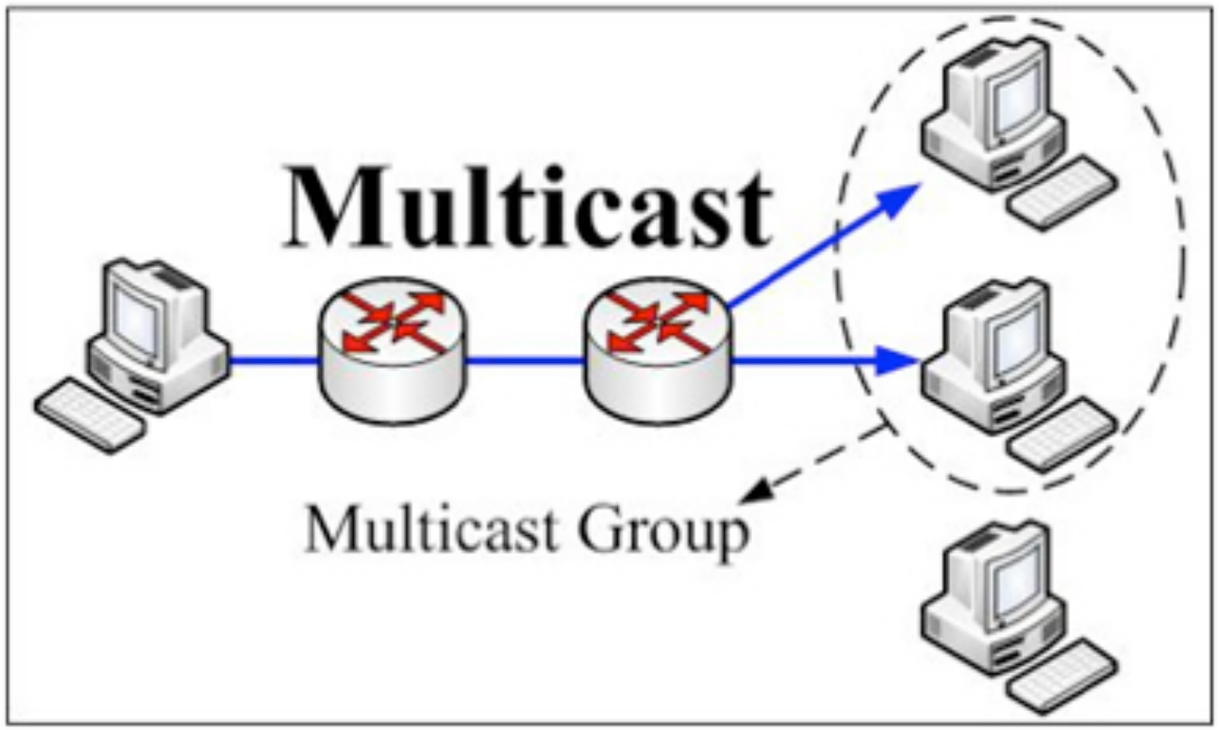
Nicola Dragoni
Embedded Systems Engineering
DTU Informatics

1. Point-to-point Communication
   - Characteristics of Interprocess Communication
   - Sockets
   - Client-Server Communication over UDP and TCP
2. Group (Multicast) Communication



Unicast VS Multicast





The Characteristics of Interprocess Communication

- Message passing between a pair of processes supported by two communication operations: send and receive

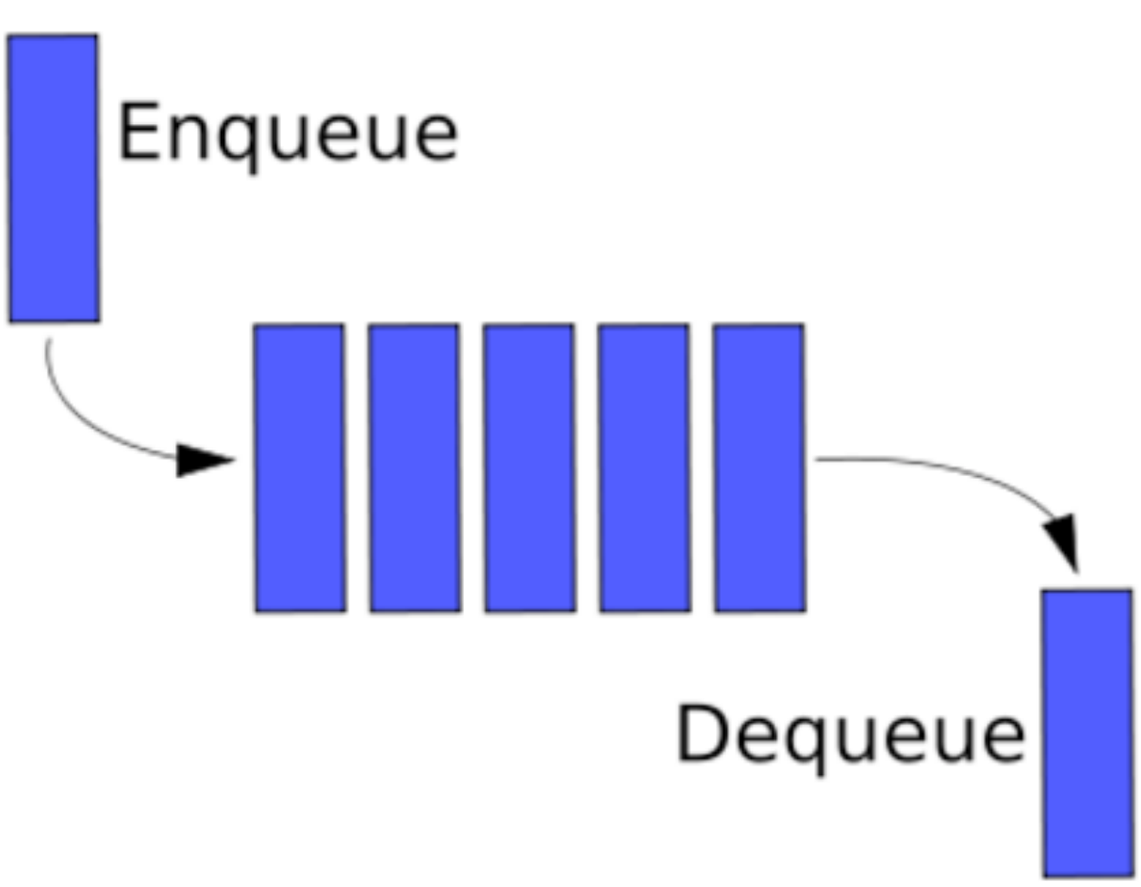
- Defined in terms of destinations and messages.

- In order for one process A to communicate with another process B:
  - A sends a message (sequence of bytes) to a destination
  - another process at the destination (B) 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 (==> CSP!).

Sending VS Receiving

- A queue is associated with each message destination.
- Sending processes cause messages to be added to *remote* queues.
- Receiving processes remove messages from *local* queues.



Communication between the sending and receiving process may be either synchronous or asynchronous.

Synchronous 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 is *blocked* until the corresponding receive is issued;

  - whenever a receive is issued the receiving process *blocks* until a message arrives.

Asynchronous Communication

- The send operation is *non-blocking*:
  - the sending process is allowed to proceed as soon as the message has been copied to a local buffer;
  - the transmission of the message proceeds in parallel with the sending process.
- The receive operation can have *blocking* and *non-blocking* variants:
  - [non-blocking] the receiving process proceeds with its program after issuing a receive operation;
  - [blocking] receiving process blocks until a message arrives.

Message Destinations?

- Usually take the form (address, local port).
  - For instance, in the Internet protocols messages are sent to (Internet address, local port) pairs.
- Local port: **message destination within a computer**, specified as an integer. It is commonly used to identify a specific service (ftp, ssh, ...).
- A port has exactly one receiver but can have many senders.
- Processes may use multiple ports from which 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.

Socket Abstraction

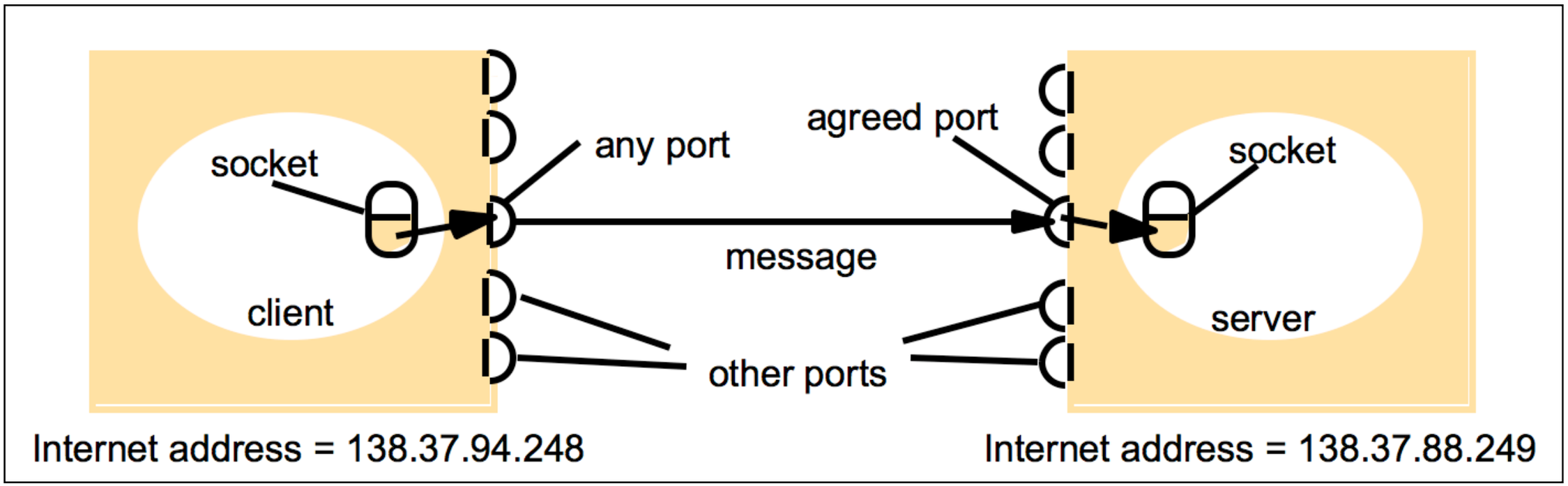
- At the programming level, message destinations can usually be defined by means of the concept of socket.

- A socket is an abstraction which provides an endpoint for communication between processes.

- A socket address is the combination of an IP address (the location of the computer) and a port (a specific service) into a single identity.

- Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process.

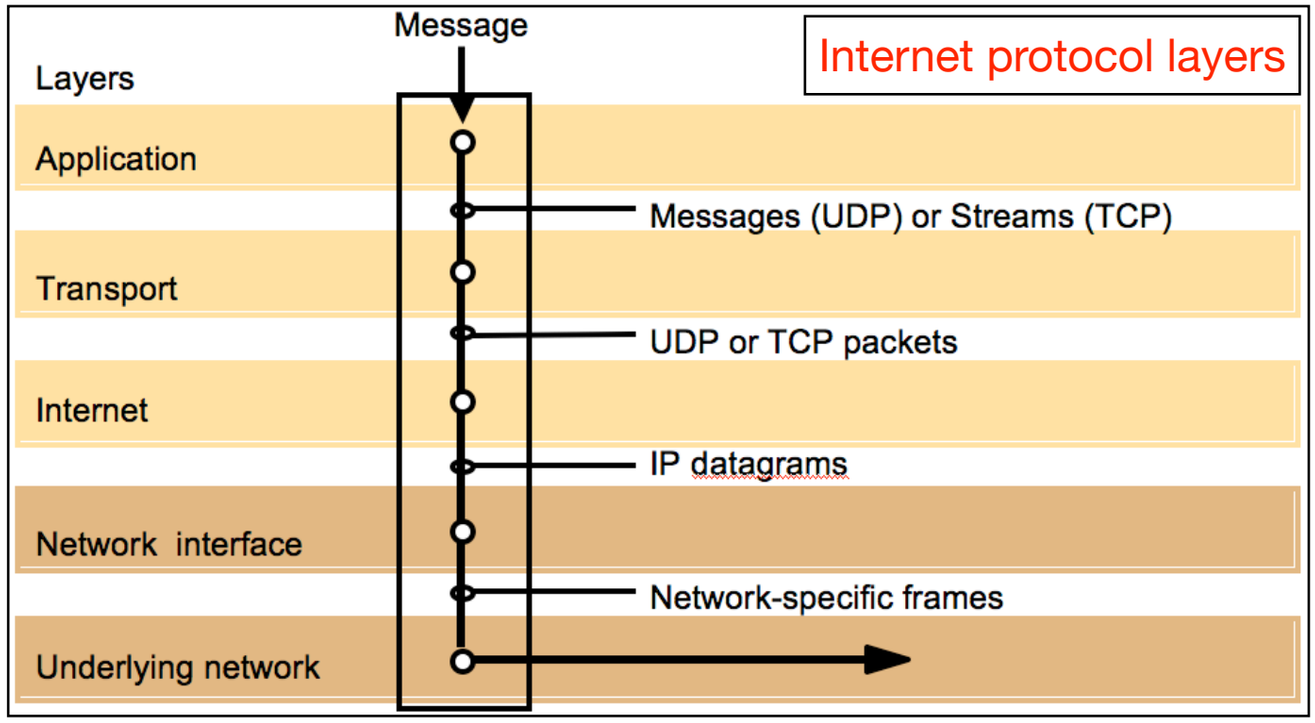
Sockets and Ports

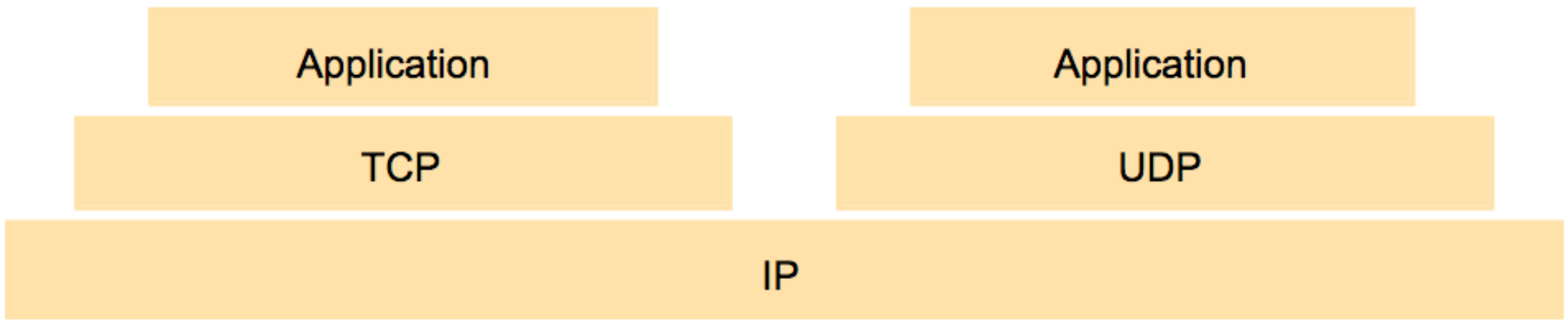


- Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number.
- Processes may use the same socket for sending and receiving messages.
- Any process may make use of multiple ports to receive messages, BUT a process cannot share ports with other processes on the same computer.
- Each socket is associated with a particular protocol, either UDP or TCP.

UDP vs TCP in a Nutshell

- TCP (Transport Control Protocol) and UDP (User Datagram Protocol) are two transport protocols.
- TCP is a reliable, connection-oriented protocol.
- UDP is a connectionless protocol that does not guarantee reliable transmission.





UDP Datagram Communication

- A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are **not** guaranteed.

- 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.

- Use of UDP: for some applications, it is acceptable to use a service that is liable to occasional omission failures.
  - DNS (Domain Name Service), which looks up DNS names in the Internet, is implemented over UDP.
  - VOIP (Voice Over IP) also runs over UDP.

Case Study: JAVA API for UDP Datagrams

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

DatagramPacket Class

- 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.

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

```
...
byte [] m = args[0].getBytes();
InetAddress aHost = InetAddress.getByName(args[1]);
int serverPort = 6789;
DatagramPacket request = new DatagramPacket(m,  args[0].length(), aHost, serverPort);
...
```

- Instances of DatagramPacket may be transmitted between processes when one process sends it and another receives it.

DatagramPacket Class

- The 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.

```
...
byte[] buffer = new byte[1000];
DatagramPacket request = new DatagramPacket(buffer, buffer.length);
...
```

- A message can be retrieved from DatagramPacket by means of the method getData.
- The methods getPort and getAddress access the port and Internet address.

```
...
aSocket.receive(request);
DatagramPacket reply = new DatagramPacket(request.getData(),
                        request.getLength(), request.getAddress(), request.getPort());
...
```

DatagramSocket Class

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

```
aSocket = new DatagramSocket(6789);
```

- It also provides a no-argument constructor that allows the system to choose a free local port.

```
aSocket = new DatagramSocket();
```

- Main methods of the class:
  - send and receive: for transmitting datagrams between a pair of sockets.
  - setSoTimeout: to set a timeout (the receive method will block for the time specified and then trow an InterruptedIOException).

Example: 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();}
   }
}
```

args[0] is a message
args[1] is a DNS name of the server

how to send a msessage

message converted in array of bytes

IP address of the host

how to receive a message

close the socket

Example: 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();}
    }
}
```

socket bound to the server port 6789

receive the msg

sends back the same message to the client

close the socket

End of the Case Study

TCP Stream Communication

- TCP is a reliable, connection-oriented protocol.
- The API for stream communication assumes that when a pair of processes are establishing a connection, one of them plays the client role and the other plays the server role, but thereafter they could be peers.
  - 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 requests connections.
  - When the server accepts 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.

TCP Stream Communication

- In other words, the API of the TCP protocol provides the abstraction of a stream of bytes to which data may be written and from which data may be read.

- The pair of sockets in client and server are connected by a pair of streams, one in each direction.
  - Thus each socket has an input stream and an output stream.
  - A process A can send information to a process B by writing to A's output stream.
  - A process B obtains the information by reading from B's input stream.

Use of TCP

- Many frequently used services run over TCP connections, with reserved port numbers, such as:
  - HTTP (HyperText Transfer Protocol, used for communication between web browsers and web servers)
  - FTP (File Transfer Protocol, it allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection)
  - Telnet (it provides access by means of a terminal session to a remote computer)
  - SMTP (Simple Mail Transfer Protocol, used to send email between computers).

Case Study: JAVA API for TCP streams

The Java API provides TCP stream communication by means of two classes: ServerSocket and Socket.

ServerSocket Class

- Used 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 of messages, or if the queue is empty, it blocks until one arrives.

- The result of executing accept is an instance of the class Socket - a socket for giving access to streams for communicating with the client.

```
int serverPort = 7896;
ServerSocket listenSocket = new ServerSocket(serverPort);
while(true) {
        Socket clientSocket = listenSocket.accept();
        EchoThread c = new EchoThread(clientSocket);
}
```

Socket Class

- The client uses a constructor to create a socket, specifying the 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.

```
...
int serverPort = 7896;
s = new Socket(args[1], serverPort);
...
```

- It can throw an UnkownHostException if the hostname is wrong or an IOException if an IO error occurs.

- The class provides methods getInputStream and getOutputStream for accessing the two streams associated with a socket.

Example: 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());}}
        }
}
```

input & output streams

socket bound to hostname and server port 7896

send message & wait for reply (write to output stream & read from input stream)

close the socket

Example: 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();
                    EchoThread c = new EchoThread(clientSocket);
                }
            } catch(IOException e) {System.out.println("Listen :" + e.getMessage());}
    }
}

// Java code continues on the next slide
```

server socket on port 7896

server listens for *connect* requests

When a *connect* request arrives, server makes a new thread in which to communicate with the client.

Example: TCP Server Makes a Connection for Each Client and Then Echoes the Client's Request

```
class EchoThread extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public EchoThread (Socket aClientSocket) {
            try {
                    clientSocket = aClientSocket;
                    in = new DataInputStream( clientSocket.getInputStream());
                    out =new DataOutputStream( clientSocket.getOutputStream());
                    this.start();
             } catch(IOException e)  {System.out.println(EchoThread:"+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*/}}
        }
}
```

socket's input and output streams

thread waits to read a msg and writes it back

close the socket

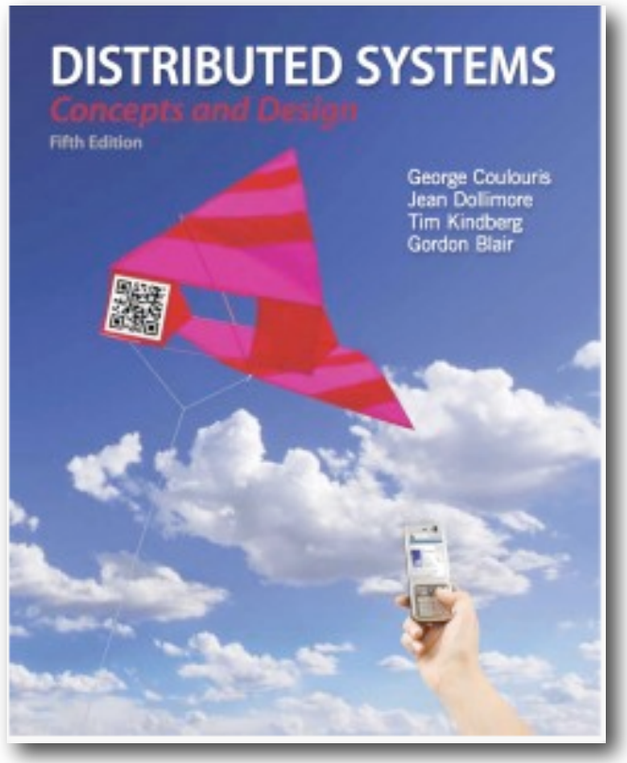
End of the Case Study

Closing a Socket

- When an application closes a socket: it will not write anymore data to its output stream.
  - Any data in the output buffer is sent to the other end of the stream and out in the queue at the destination socket with an indication that the stream is broken.
- When a process has closed its socket, it will no longer able to use its input and output streams.
- The process at the destination can read the data in its queue, but any further reads after the queue is empty will result in an error/exception (for instance, EOFException in Java).
- When a process exits or fails, all of its sockets are eventually closed.
- Attempts to use a closed socket or to write to a broken stream results in an error/exception (for instance, IOException in Java).

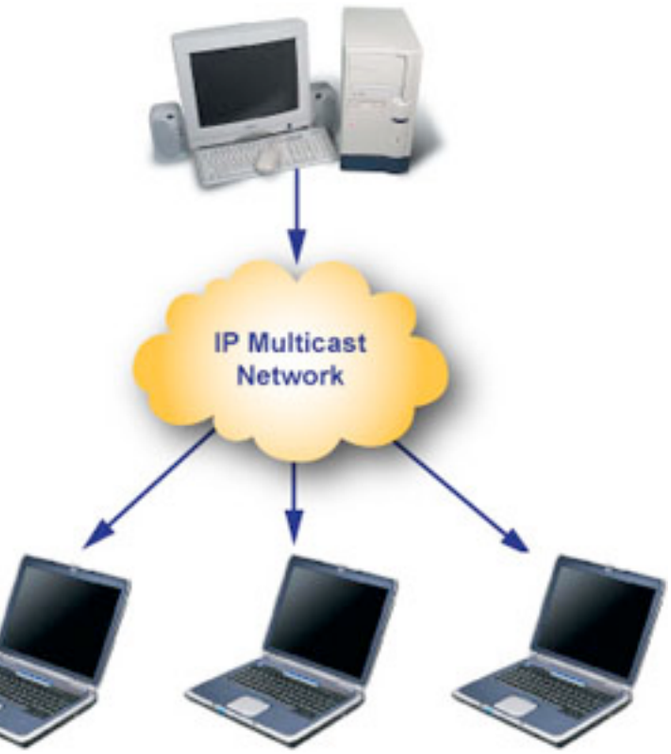
Interprocess Communication

1. Point-to-point Communication
   - Characteristics of Interprocess Communication
   - Sockets
   - Client-Server Communication over UDP and TCP
2. Group (Multicast) Communication



Multicast

- A multicast operation 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 provides no guarantees about message delivery or ordering.

What Can Multicast Be Useful for?

- Multicast messages provides 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 members.
   - 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.

What Can Multicast Be Useful for?

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

2. 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.

What Can Multicast Be Useful for?

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

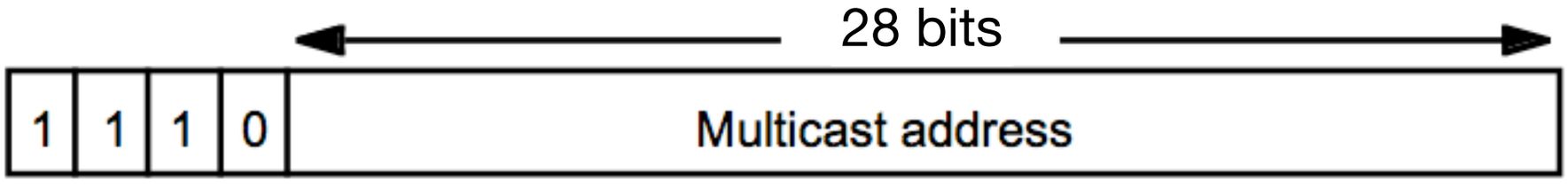
3. Propagation of event notifications

   - Multicast to a group may be used to notify processes when sometimes happens.

   - For example, a news system might notify interested users when a new message has been posted on a particular newsgroup.

IP Multicast

- IP multicast is built on top of the Internet Protocol, IP.
- Note that IP packets are addressed to computers - ports belong to the TCP and UDP levels.
- 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 an Internet address whose first 4 bits are 1110 (in IPv4).



IP Multicast - Membership

- Being a member of a multicast group allows a computer to receive IP packets sent to the group.

- 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.

IP Multicast - Programming Level

- 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.
  - An application program can join a multicast group by making its socket join the group, enabling it to receive messages to the group.

IP Multicast - IP Level

- At the IP level:
  - A computer belongs to a multicast group when one or more of its processes has sockets that belong to that 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.

Case Study: JAVA API for IP Multicast

The Java API provides a datagram interface to IP multicast through the class MulticastSocket.

The Class MulticastSocket

- A subclass of DatagramSocket with the additional capability of being able to join multicast groups.

- It provides two alternative constructors, allowing sockets to be created to use either a specified local port or any free local port.

```
...
MulticastSocket s =null;
s = new MulticastSocket(6789);
...
```

Joining a Group

- A process can **join** a group with a given multicast address by invoking the joinGroup method of its multicast socket.
  - In this way, 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*.

```
...
MulticastSocket s =null;
s = new MulticastSocket(6789);
InetAddress group = InetAddress.getByName(args[1]);
s.joinGroup(group);
...
```

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

Leaving a Group

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

```
...
MulticastSocket s =null;
InetAddress group = InetAddress.getByName(args[1]);
s = new MulticastSocket(6789);
...
s.leaveGroup(group);
```

Example:
Multicast Peer Joins a Group and Sends and Receives Datagrams

```
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);


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

args[0] = msg contents
args[1] = multicast address

MulticastSocket creation on port 6789

joining a multicast group

sending a DatagramPacket message

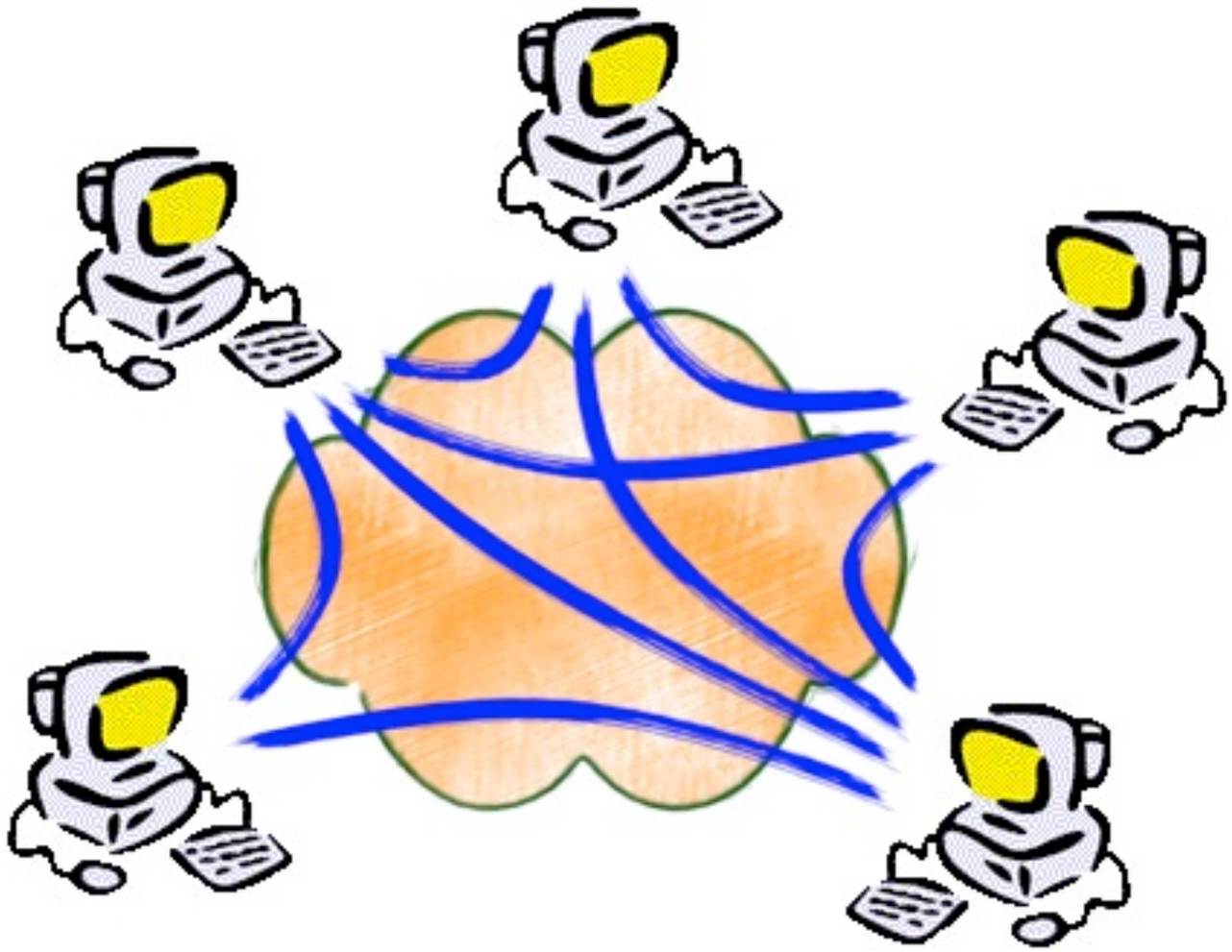
Example:
Multicast Peer Joins a Group and Sends and Receives Datagrams

peer attempts to receive 3 multicast messages from its peers via its socket

```
            // get messages from others in group
                    byte[] buffer = new byte[1000];
                    for(int i=0; i< 3; i++) {
                        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();}
    }
}
```

Example:
Multicast Peer Joins a Group and Sends and Receives Datagrams

- 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 that joined after it.



End of the Case Study