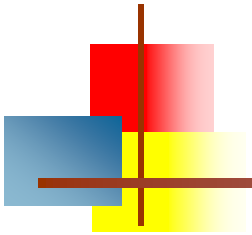
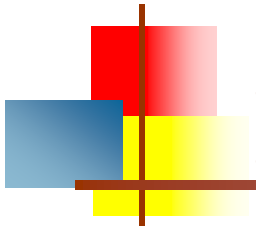


Interprocess Communication



Multicast communication

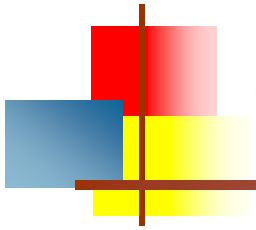
- A multicast sends a single message from one process to each of the members of a group of processes.
- The sender is unaware of the identities of the individual recipients and of the size of the group.
- IP multicast is built on top of the Internet Protocol (IP).
- A multicast group is specified by a Class D Internet address, an address whose first 4 bits are 1110 in IPv4.
- The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups.

Multicast communication

- At the application programming level, IP multicast is available only via UDP
- Multicast routers. IP packets can be multicast both on a local network and on the wider Internet.

EX:

- 224.0.0.1 (all-systems.mcast.net) – all systems on the local subnet.
- - 224.0.0.2 (all-routers.mcast.net) – all routers on the local subnet.
- - 224.0.0.11 (mobile-agents.mcast.net) – mobile agents on the local subnet



Multicast communication

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

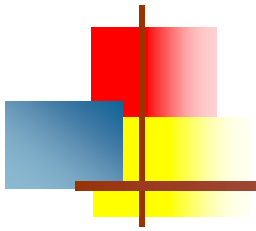
- A process can join a multicast group with a given multicast address by invoking the joinGroup method of its multicast socket.

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

Multicast peer joins a group and sends and receives datagrams

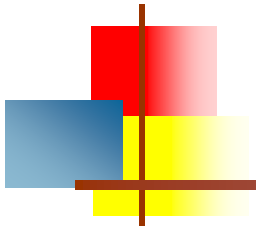
```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
         try {
                InetAddress group = InetAddress.getByName(args[1]);
                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
```



continued

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

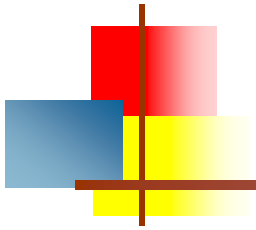


Transmission Control Protocol (TCP)

- Connection oriented
  - Explicit set-up and tear-down of TCP session
- Stream-of-bytes service
  - Sends and receives a stream of bytes, not messages
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Acknowledgments & retransmissions for reliable delivery
  - Sequence numbers to detect losses and reorder data
- Flow control
  - Prevent overflow of the receiver's buffer space
- Congestion control
  - Adapt to network congestion for the greater good

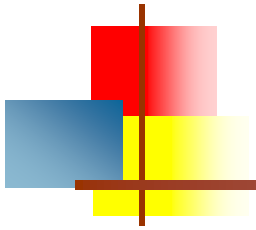
An Analogy: Talking on a Cell Phone

- Alice and Bob on their cell phones
  - Both Alice and Bob are talking
- What if Alice couldn't understand Bob?
  - Bob asks Alice to repeat what she said
- What if Bob hasn't heard Alice for a while?
  - Is Alice just being quiet?
  - Or, have Bob and Alice lost reception?
  - How long should Bob just keep on talking?
  - Maybe Alice should periodically say "uh huh"
  - … or Bob should ask "Can you hear me now?" ☺



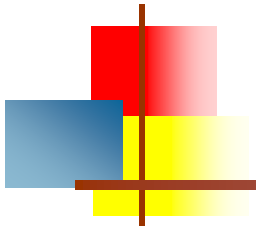
Some Take-Aways from the Example

- Acknowledgments from receiver
  - Positive: "okay" or "ACK"
  - Negative: "please repeat that" or "NACK"
- Timeout by the sender ("stop and wait")
  - Don't wait indefinitely without receiving some response
  - … whether a positive or a negative acknowledgment
- Retransmission by the sender
  - After receiving a "NACK" from the receiver
  - After receiving no feedback from the receiver



Challenges of Reliable Data Transfer

- Over a perfectly reliable channel
  - All of the data arrives in order, just as it was sent
  - Simple: sender sends data, and receiver receives data
- Over a channel with bit errors
  - All of the data arrives in order, but some bits corrupted
  - Receiver detects errors and says "please repeat that"
  - Sender retransmits the data that were corrupted
- Over a lossy channel with bit errors
  - Some data are missing, and some bits are corrupted
  - Receiver detects errors but cannot always detect loss
  - Sender must wait for acknowledgment ("ACK" or "OK")
  - … and retransmit data after some time if no ACK arrives



TCP Support for Reliable Delivery

- Checksum
  - Used to detect corrupted data at the receiver
  - …leading the receiver to drop the packet
- Sequence numbers
  - Used to detect missing data
  - ... and for putting the data back in order
- Retransmission
  - Sender retransmits lost or corrupted data
  - Timeout based on estimates of round-trip time
  - Fast retransmit algorithm for rapid retransmission

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());}
   }
}
```

// this figure continues on the next slide