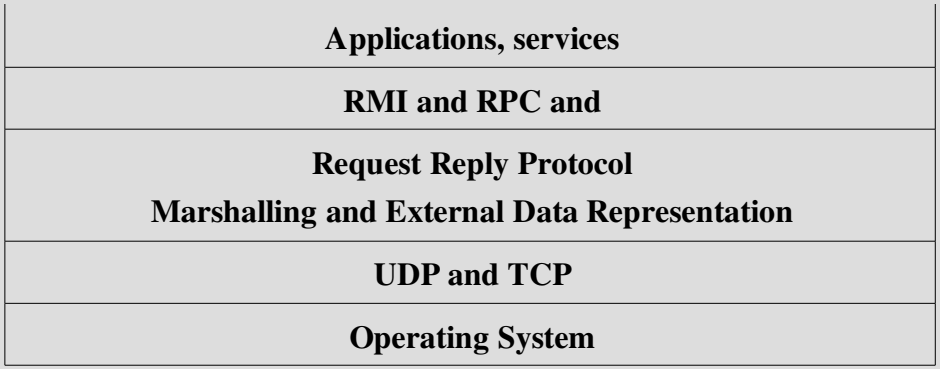
Interprocess Communication (IPC)

The characteristics of protocols for communication between processes in a distributed system

Exploring the Middleware Layers



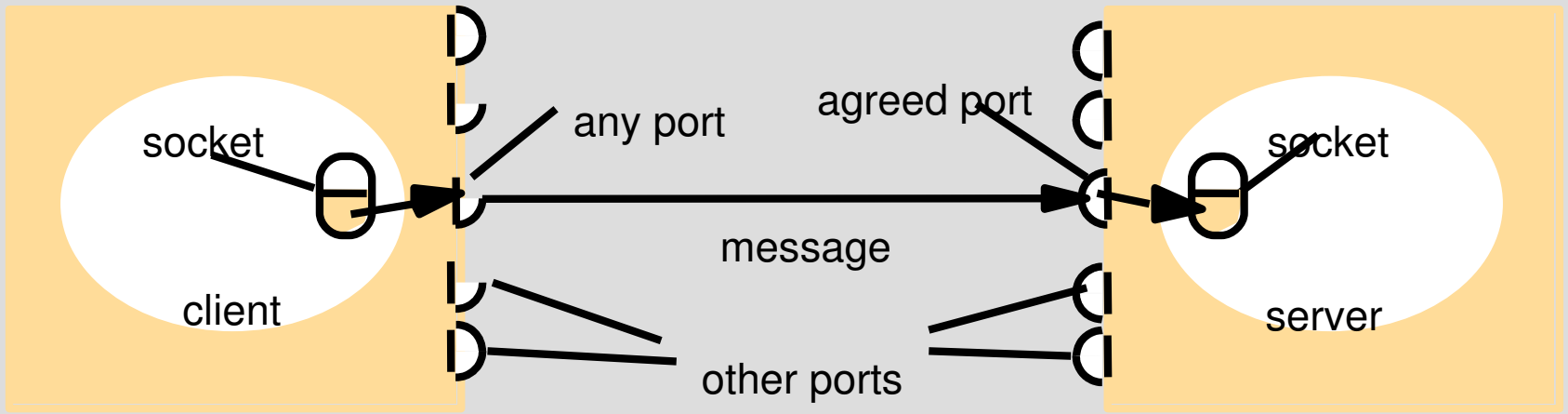
Characteristics of IPC

- Message passing between a pair of processes supported by SEND and RECEIVE operations
- Synchronous - sending and receiving processes synchronize every message, and BLOCK
- Asynchronous - sending is NON-BLOCKING, receiving can be both BLOCKING and NON-BLOCKING
- Non-blocking receives are complex, so most systems employ the blocking form of receive

Other IPC Characteristics

- Message destinations - typically specified as address/port pairs (end-points)
- Reliability - both reliable and unreliable IPCs are possible
- Ordering - often, applications require SENDER ORDERING to be maintained

Example IPC Mechanism - Sockets



Internet address = 138.37.94.248

Internet address = 138.37.88.249

UDP Datagram Communication

- Datagrams sent without ACKs or retries
- Message sizes are often pre-negotiated
- Fragmentation can occur
- Blocking sends and receives are common - timeouts can be used, but these can be tricky
- Datagram discarding occurs when no receiving process is waiting

UDP's Failure Model

- Omission Failures - messages dropped, checksum errors, lack of buffer space
- Both send-omissions and receive-omissions can occur
- Ordering - messages can arrive out-of-order
- Applications that use UDP need to provide their own checks

Usages of UDP

- Applications that do not suffer from the overheads associated with guaranteed message delivery
- DNS
- VoIP

Example UDP Client in Java

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

Example UDP Server in Java

```
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 Streamed Communication

- Stream of bytes transferred from sender to receiver
- Characteristics of the network are hidden/transparent to applications
- Messages sizes can be small or large
- An ACK scheme deals with lost messages
- Flow control mechanisms throttle fast senders
- Message duplication is handled, ordering is maintained
- Message destinations are "stream end-points"

More of TCP

- When establishing communication, one side is the client, the other is the server
- Thereafter, both can operate as peers, if needs be
- Pairs of sockets are connected by pairs of streams, one for input, the other for output

TCP's Failure Model

- Checksums detect and reject corrupt packets
- Sequence numbers detect and reject duplicate packets
- Timeouts and retransmissions deal with lost packets
- TCP is not totally reliable, as it does not guarantee delivery of messages in the face of all possible difficulties

TCP's Unreliability

- When a connection is broken, a process is notified if it attempts to read or write
- Has the network failed or has the process at the other end-point failed?
- Where are previous sent messages actually received?

Example TCP Client in Java

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

Example TCP Server in Java

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

IPC and Data

External Data Representation and Marshalling

The Problem

- Running programs (processes) are represented as (binary) data structures
- Information in messages is represented as a sequence of bytes
- How do we transform one into the other and vice-versa?

Flattening

- Data structures must be flattened into a sequence of bytes before transmission and rebuilt on receipt
- Byte-ordering (little- or big-endian?) is an issue
- Character encodings (ASCII, Unicode) are an issue, too

Exchanging Binary Data

- Values are converted to an agreed external format
- Values are transmitted in the sender's format; the recipient converts that values if necessary
- An agreed standard for the representation of data structures and primitive values is called an "external data representation"

Marshalling and Unmarshalling

- Marshalling - taking a collection of data items and assembling them into a form suitable for transmission in a message
- Unmarshalling - disassembling a message on arrival to produce an equivalent collection of data items at the destination

Three Alternative Approaches

- CORBA's Common Data Representation (CDR) - can be used with a variety of programming technologies
- Java's Object Serialization - works only within the Java environment
- XML (Extensible Markup Language) - a textual format for representing structured data that works with any programming technology

CORBA's CDR

- CDR can represent 15 primitive types and a range of composite types
- Both little- and big-endian support is provided - senders indicate in which ordering the message is transmitted
- Floating-point numbers use the IEEE standard
- Characters are represented in a code-set agreed between the sender and receiver
- Data type information is NOT transmitted

CORBA CDR's Composite Types

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

CORBA CDR - Example Message

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*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1934}

Marshalling in CORBA

- CORBA's Interface Definition Language (IDL) is used to "automatically" produce marshalling and unmarshalling operations
- The CORBA IDL compiler enables the generation of the required components

Example CORBA IDL

```
struct Person
{
        string name;
        string place;
        unsigned long year;
};
```

Java's Object Serialization

- The term "serialization" refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message
- Consider this code :

```
Person p = new Person( "Smith", "London", 1934 );
```

Serialized Form of "p"

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*

The true serialized form contains additional type markers; h0 and h1 are handles

Extensible Markup Language (XML)

- A "markup language" refers to a textual encoding that represents both a text and details as to its structure or its appearance
- HTML was designed to describe the appearance of web pages
- XML was designed to describe structured documents and markup languages

XML Characteristics

- XML is "extensible" in the sense that users can define their own tags
- XML is "self-describing"
- XML was intended to be used by multiple applications for different purposes
- XML is "textual", so can be easily read by humans and computers

Example XML (Elements and Attributes)

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1934</year>
        <!-- a comment -->
</person >
```

More XML

- The names used in XML are user-defined and follow the normal naming conventions
- Binary data is (typically) represented in "base64"

XML Parsing and Well Formed Documents

- Every start-tag has a matching end-tag
- All tags are nested correctly
- All XML documents have a single root element within which all other elements are enclosed
- The CDATA notation allows for the inclusion of special characters

XML Prologs

```
<?XML version="1.0" encoding="UTF-8"
        standalone="yes" ?>
```

XML Namespaces

- A set of names for a collection of element types and attributes
- The namespace convention allows an application to make use of multiple sets of external definitions in different namespaces without the risk of name clashes

Example XML Namespace

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1934 </pers:year>
</person>
```

XML Schemas

- Defines the elements and attributes that can appear in a document
- Defines how the elements are nested, the order and number of elements
- Defines whether or not an element is empty or can include text
- For each element, the schema defines the type and default value

XML Schema Example

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions    >
   <xsd:element name= "person" type ="personType" />
      <xsd:complexType name="personType">
         <xsd:sequence>
            <xsd:element name = "name"  type="xs:string"/>
            <xsd:element name = "place"  type="xs:string"/>
            <xsd:element name = "year"  type="xs:positiveInteger"/>
         </xsd:sequence>
         <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
      </xsd:complexType>
</xsd:schema>
```

Valid XML Documents

An XML document that is defined to conform to a particular schema may also be validated by means of that schema (using one of the many programming APIs)

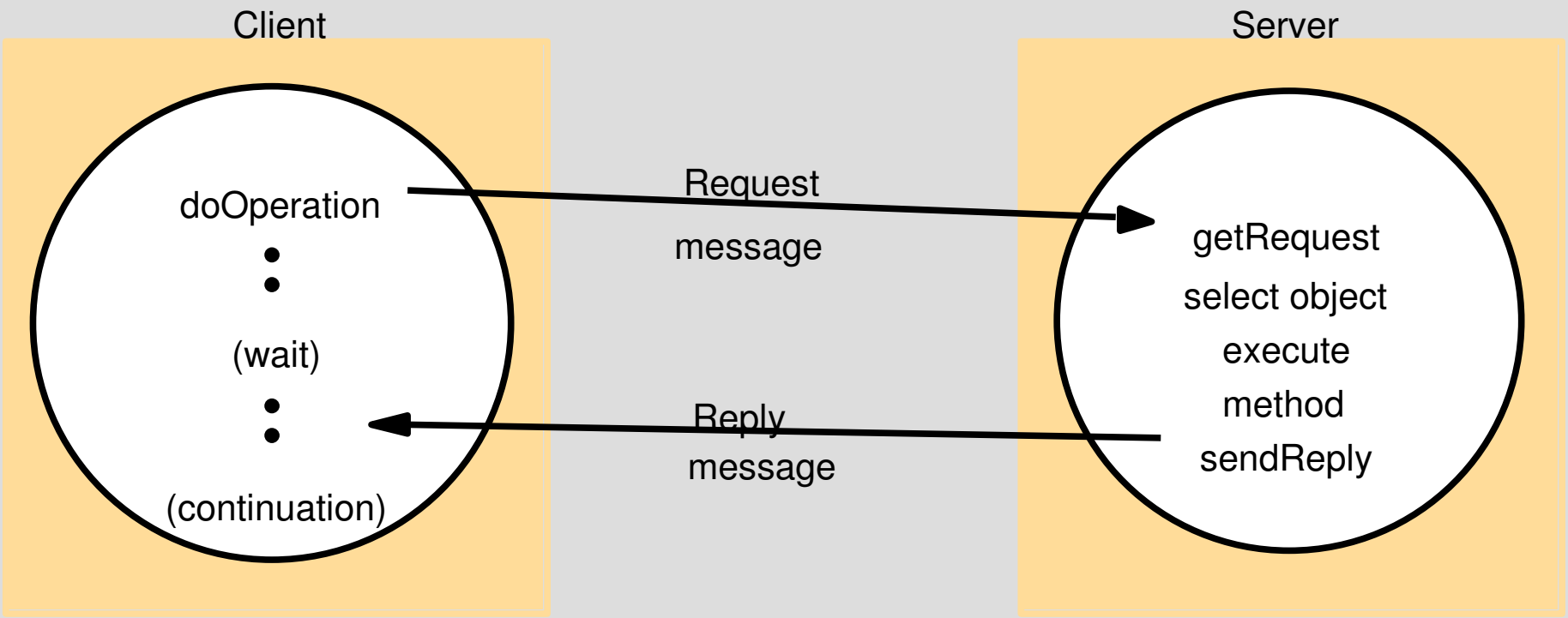
Client-Server Communication

- Normally, request-reply communication is synchronous because the client process blocks until the reply arrives from the server
- It can also be reliable as the reply from the server is effectively an acknowledgment to the client
- It is possible to build a client-server protocol over a reliable or unreliable protocol

Avoiding Unnecessary Overhead

- ACKs are unnecessary when requests are followed by replies
- Establishing a connection involves (at least) two extra pairs of messages in addition to the request-reply messages
- Flow control is overkill, as most invocations pass only small arguments and results

Request-Reply Communications



The Request-Reply 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.

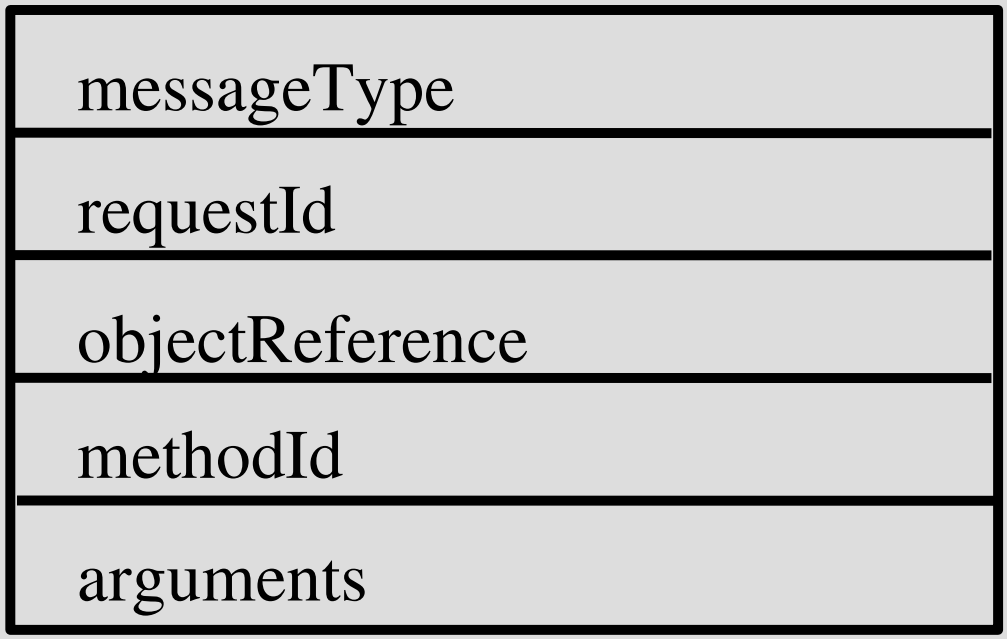
public byte[] getRequest ();

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.

The Request-Reply Message Structure



int (0=Request, 1= Reply)

int

RemoteObjectRef

int or Method

array of bytes

Request-Reply Communication Characteristics

- What is the failure model? (Can omissions occur? Is message ordering maintained?)
- Are timeouts employed on operations?
- How are duplicate request messages handled?
- How are lost reply messages handled?
- Is a history of requests (and replies) maintained on either end?

Idempotent Operations

An operation is **idempotent** if it can be executed one or more times without side-effects

An Example Request-Reply Protocol - HTTP

- Allows for the invocation of methods on web resources
- Content negotiation is also supported
- Password-style authentication is available

How HTTP Works

- Implemented over TCP
- Initially employed a simple Connect-Request-Reply-Close cycle
- This proved to be expensive and inefficient
- Latest version of HTTP supports "persistent connections"

HTTP Requests and Replies

- R'n'Rs are marshalled into ASCII strings
- Resources can be byte sequences and may be compressed
- Multipurpose Internet Mail Extensions (MIME) supports multi-part messages of varying formats

HTTP Methods

- GET (which is generally idempotent)
- HEAD, POST, PUT, DELETE, OPTIONS and TRACE (are the most widely supported)

HTTP Request and Reply Messages

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

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

Group Communication

The pairwise exchange of messages is rarely the best model for communication from one process to a group of other processes

Group Communication - Multicasting

- The membership of the group is transparent to the sender
- Multicast messages provide a useful infrastructure for constructing distributed systems

Uses of Multicasting

- Fault tolerance based on replicated servers
- Finding the discovery service in spontaneous networking
- Better performance through replicated data
- Propagation of event notifications

Group Communications with IP Multicasting

- IP Multicast is built on top of IP
- The sender transmits a single IP packet to a set of computers that form a multicast group
- The sender does not know the recipients identities nor how big the group is
- The class D address space within IP is reserved for IP Multicast

Characteristics of IP Multicast

- Available with UDP only
- Identified by an IP address/port-number “end-point”
- Applications can join a multicast group by opening a socket to the end-point
- Multicast address range - 224.0.0.1 through 224.0.0.255

IP Multicast's Failure Model

- Same as for UDP datagrams
- Multicasts suffer from omission failures
- Not all of the group members receive everything
- Reliable multicasting is possible - overheads are high

Example Multicast Peer in Java

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

Multicasting Reliability and Ordering

- Suffers from omission failures!
- Recipients may drop messages due to full buffers
- A datagram lost at one multicast router prevents all those routers beyond from receiving the datagram
- A multicast router can fail
- Message ordering "errors" can result in two routers receiving a sequence of multicasts in a very different order to that which was sent

Cast Study - UNIX IPC

The Socket system calls layered over the Internet TCP and UDP protocols

Socket Datagram Communications

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

ServerAddress and *ClientAddress* are socket addresses

Socket Stream Communications

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
bind(s, ServerAddress);
listen(s,5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

ServerAddress and *ClientAddress* are socket addresses

IPC Summary - Exchange Protocols

- The request (R) protocol - no value need be returned to the client
- The request-reply (RR) protocol - special ACKs are not required, the reply and subsequent new requests suffice as ACKs
- The request-reply-ack-reply (RRA) protocol - used when the server maintains a history of messages; the "ack-reply" allows the server to remove items from its history