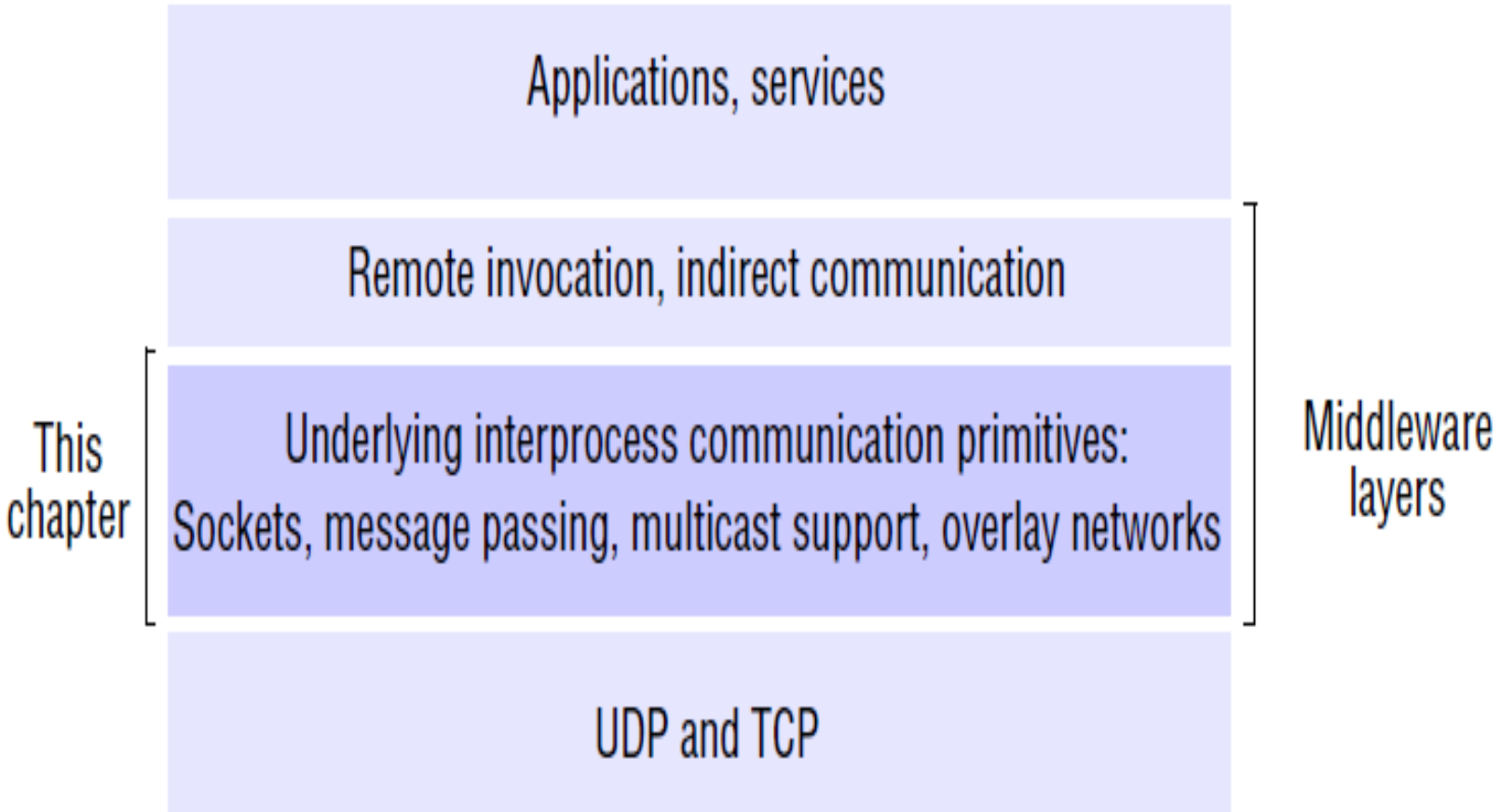
CHAPTER -4
INTERPROCESS COMMUNICATION

Figure 4.1 Middleware layers



Introduction

- Interprocess communication in the Internet provides both datagram and stream communication.
- The interprocess communication primitives support point-to-point communication, yet it is equally useful to be able to send a message from one sender to a group of receivers.
- Multicast is an important requirement for distributed applications and must be provided even if underlying support for IP multicast is not available. This is typically provided by an overlay network constructed on top of the underlying TCP/IP network.
- Overlay networks can also provide support for file sharing, enhanced reliability and content distribution.

Introduction (cont.)

- In this chapter we will see how middleware and application programs can use transport level protocols UDP and TCP.
- The application program interface to UDP provides a *message passing* abstraction – the simplest form of interprocess communication. This enables a sending process to transmit a single message to a receiving process.
- The independent packets containing these messages are called *datagrams*. In the Java and UNIX APIs, the sender specifies the destination using a socket – an indirect reference to a particular port used by the destination process at a destination computer.
- The application program interface 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.

Introduction (cont.)

- Streams provide a building block for producer-consumer communication.
- A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them.
- The data items sent by the producer to the consumer are queued on arrival at the receiving host until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used to hold the queued data items is exhausted.

The API for the Internet Protocols

- We will discuss the general characteristics of interprocess communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.
- This section revisits the message communication operations *send* and *receive*, with a discussion of how they synchronize with one another and how message destinations are specified in a distributed system. The next section introduces *sockets*, which are used in the application programming interface to UDP and TCP. Thereafter, another section discusses UDP and its API in Java. Finally, TCP and its API in Java will be discussed. The APIs for Java are object oriented but are similar to the ones designed originally in the Berkeley BSD 4.x UNIX operating system; a case study on the latter is available on the web site for the book [www.cdk5.net/ipc]. Readers studying the programming examples in this section should consult the online Java documentation or Flanagan [2002] for the full specification of the classes discussed, which are in the package *java.net*.

Characteristics of Interprocess Communication

- Message passing between a pair of processes can be supported by two message communication operations, *send* and *receive*, defined in terms of destinations and messages.
- To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination 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.

Synchronous and Asynchronous Communication

- A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving processes may be either synchronous or asynchronous.
- In the ***synchronous*** form of 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 (or thread) is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued by a process (or thread), it blocks until a message arrives.

Synchronous and Asynchronous Communication (cont.)

- In the ***asynchronous*** form of communication, the **use of the *send* operation is *non-blocking*** in that the **sending process** is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process.
- The ***receive* operation can have blocking and non-blocking** variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

Synchronous and Asynchronous Communication (cont.)

- In a system environment such as Java, which supports multiple threads in a single process, the blocking *receive* has no disadvantages, for it can be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage.
- Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control.
- For these reasons, today's systems do not generally provide the non-blocking form of *receive*.

Message Destinations

- Previous chapter explained that in the Internet protocols, messages are sent to (*Internet address*, *local port*) pairs. A local port is a message destination within a computer, specified as an integer.
- A port has exactly one receiver (multicast ports are an exception) but can have many senders. Processes may use multiple ports 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.
- If the client uses a fixed Internet address to refer to a service, then that service must always run on the same computer for its address to remain valid.

Message Destinations (cont.)

- This can be avoided by using the following approach to providing location transparency:

  - Client programs refer to services by name and use a name server or to translate their names into server locations at runtime. This allows services to be relocated but not to migrate – that is, to be moved while the system is running.

Reliability

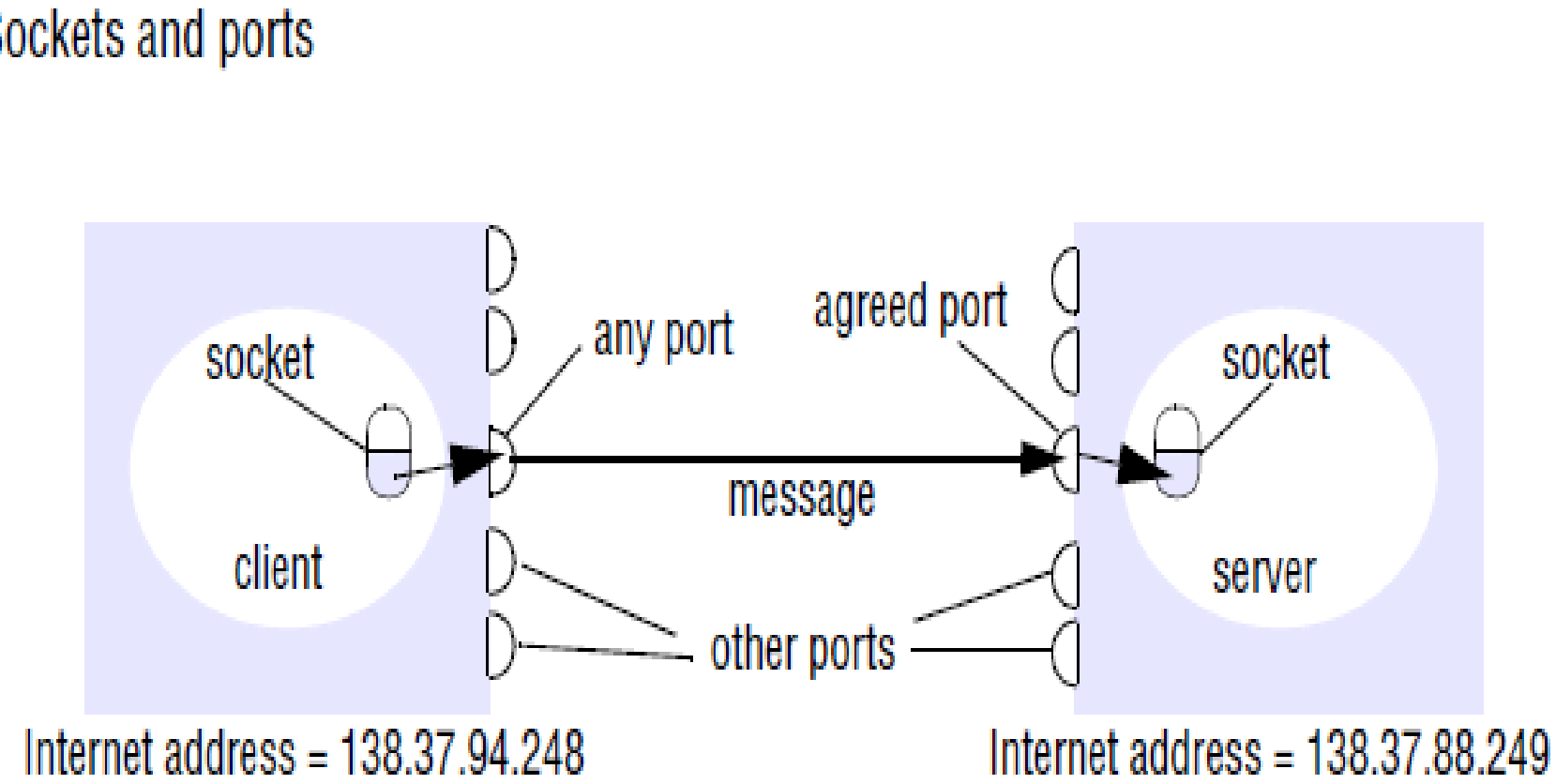
- As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a ‘reasonable’ number of packets being dropped or lost.
- In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost.
- For integrity, messages must arrive uncorrupted and without duplication.

Ordering

- Some applications require that messages be delivered in *sender order*– that is, the order in which they were transmitted by the sender.
- The delivery of messages out of sender order is regarded as a failure by such applications.

Sockets

- Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes.



Sockets (cont.)

- For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs.
- 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. Each computer has a large number (2^{16}) of possible port numbers for use by local processes for 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. (Processes using IP multicast are an exception in that they do share ports.)
- However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

Java API for Internet Addresses

- As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses. Users of this class refer to computers by Domain Name System (DNS) hostnames. For example, instances of *InetAddress* that contain Internet addresses can be created by calling a static method of *InetAddress*, giving a DNS hostname as the argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is *bruno.dcs.qmul.ac.uk*, use:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");

- This method can throw an *UnknownHostException*. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet addresses – 4 bytes in IPv4 and 16 bytes in IPv6.

UDP Datagram Communication

- 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.
- To send or receive messages, a process must first create a socket bound to an Internet address of the local host and a local port. A server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it.
- A client binds its socket to any free local port. The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

Issues in UDP Datagram Communication (cont.)

***Message size*:**

- The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival.
- The underlying IP protocol allows packet lengths of up to 216 bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size.
- Generally, an application, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

Issues in UDP Datagram Communication (cont.)

Blocking:

- Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication. The *send* operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination.
- On arrival, the message is placed in a queue for the socket that is bound to the destination port. The message can be collected from the queue by an outstanding or future invocation of *receive* on that socket. Messages are discarded at the destination if no process already has a socket bound to the destination port. The method *receive* blocks until a datagram is received, unless a timeout has been set on the socket. If the process that invokes the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread. Threads are discussed in Chapter 7. For example, when a server receives a message from a client, the message may specify work to do, in which case the server will use separate threads to do the work and to wait for messages from other clients.

Issues in UDP Datagram Communication (cont.)

***Timeouts*:**

The *receive* that blocks forever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have been lost.

To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

Issues in UDP Datagram Communication (cont.)

***Receive from any*:**

- The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin.
- The *receive* method returns the Internet address and local port of the sender, allowing the recipient to check where the message came from.
- It is possible to connect a datagram socket to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

Failure Model for UDP Datagrams

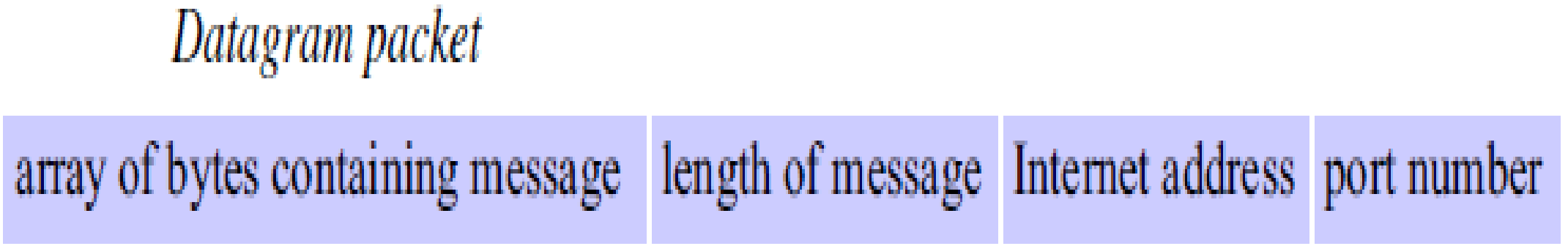
- A failure model for communication channels and defines reliable communication in terms of two properties: integrity and validity.
- The integrity property requires that messages should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams suffer from the following failures:
  - ***Omission failures*:** Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination. To simplify the discussion, we regard send-omission and receive-omission failures as omission failures in the communication channel.
  - ***Ordering*:** Messages can sometimes be delivered out of sender order.
- Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements.

Use of UDP

- For some applications, it is acceptable to use a service that is liable to occasional omission failures.
- For example, the Domain Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs over UDP.
- UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:
  - the need to store state information at the source and destination;
  - the transmission of extra messages;
  - latency for the sender.

Java API for UDP Datagrams

- The Java API provides datagram communication by means of two classes: *DatagramPacket* and *DatagramSocket*.
- ***DatagramPacket*:** 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, as follows:



Java API for UDP Datagrams- DatagramPacket (cont.)

- An instance of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it.
- This 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.
- A received message is put in the *DatagramPacket* together with its length and the Internet address and port of the sending socket. The message can be retrieved from the *DatagramPacket* by means of the method *getData.*
- The methods *getPort* and *getAddress* access the port and Internet address.

Java API for UDP Datagrams- DatagramSocket

- This class supports sockets for sending and receiving UDP datagrams.
- It provides a constructor that takes a port number as its argument, for use by processes that need to use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port.
- These constructors can throw a *SocketException* if the chosen port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.
- The class *DatagramSocket* provides methods that include the following:
- ***send* and *receive*:** These methods are for transmitting datagrams between a pair of sockets. The argument of *send* is an instance of *DatagramPacket* containing a message and its destination. The argument of *receive* is an empty *DatagramPacket* in which to put the message, its length and its origin. The methods *send* and *receive* can throw *IOExceptions*.

Java API for UDP Datagrams- DatagramSocket (cont.)

- ***setSoTimeout*:** This method allows a timeout to be set. With a timeout set, the *receive* method will block for the time specified and then throw an *InterruptedIOException*.
- ***connect*:** This method is used for connecting to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

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,  m.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 stream communication

- The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:
- ***Message sizes*:** The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.

TCP stream communication (cont.)

- ***Lost messages*:** The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals.
- If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 2006] cuts down on the number of acknowledgement messages required.
- ***Flow control*:** The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.

TCP stream communication (cont.)

- ***Message duplication and ordering*:** Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.
- ***Message destinations*:** A pair of communicating processes establish a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a *connect* request from client to server followed by an *accept* request from server to client before any communication can take place. This could be a considerable overhead for a single client-server request and reply.

TCP stream communication (cont.)

- 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 request connections. The listening socket maintains a queue of incoming connection requests. In the socket model, when the server *accept*s 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 (cont.)

- The pair of sockets in the 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.
- One of the pair of processes can send information to the other by writing to its output stream, and the other process obtains the information by reading from its input stream.
- When an application *closes* a socket, this indicates that it will not write any more data to its output stream. Any data in the output buffer is sent to the other end of the stream and put in the queue at the destination socket, with an indication that the stream is broken. The process at the destination can read the data in the queue, but any further reads after the queue is empty will result in an indication of end of stream. When a process exits or fails, all of its sockets are eventually closed and any process attempting to communicate with it will discover that its connection has been broken.

TCP stream communication (cont.)

- The following are some outstanding issues related to stream communication:
- ***Matching of data items*:** Two communicating processes need to agree as to the contents of the data transmitted over a stream.
- For example, if one process writes an *int* followed by a *double* to a stream, then the reader at the other end must read an *int* followed by a *double.*
- When a pair of processes do not cooperate correctly in their use of a stream, the reading process may experience errors when interpreting the data or may block due to insufficient data in the stream.

TCP stream communication- related issues (cont.)

- ***Blocking*:** The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available. The process that writes data to a stream may be blocked by the TCP flow-control mechanism if the socket at the other end is queuing as much data as the protocol allows.
- ***Threads*:** When a server accepts a connection, it generally creates a new thread in which to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients. In an environment in which threads are not provided, an alternative is to test whether input is available from a stream before attempting to read it; for example, in a UNIX environment the *select* system call may be used for this purpose.

Failure model for TCP stream communication

- To satisfy the integrity property of reliable communication, TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets.
- For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets. Therefore, messages are guaranteed to be delivered even when some of the underlying packets are lost.
- But if the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

Failure model for TCP stream communication (cont.)

- When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:
  - The processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection.
  - The communicating processes cannot tell whether the messages they have sent recently have been received or not.

Use of TCP

- Many frequently used services run over TCP connections, with reserved port numbers. These include the following:
  - ***HTTP*:** The Hypertext Transfer Protocol is used for communication between web browsers and web servers.
  - ***FTP*:** The File Transfer Protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.
  - ***Telnet*:** Telnet provides access by means of a terminal session to a remote computer.
  - ***SMTP*:** The Simple Mail Transfer Protocol is used to send mail between computers.

Java API for TCP streams

- The Java interface to TCP streams is provided in the classes *ServerSocket* and *Socket*:
- ***ServerSocket*:** This class is intended for use 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 or, if the queue is empty, blocks until one arrives. The result of executing *accept* is an instance of *Socket* – a socket to use for communicating with the client.
- *Socket*: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS 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. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

Java API for TCP streams (cont.)

- The *Socket* class provides the methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket.
- The return types of these methods are *InputStream* and *OutputStream*, respectively – abstract classes that define methods for reading and writing bytes. The return values can be used as the arguments of constructors for suitable input and output streams. Our example uses *DataInputStream* and *DataOutputStream*, which allow binary representations of primitive data types to be read and written in a machine-independent manner.
- Figure shows a client program in which the arguments of the *main* method supply a message and the DNS hostname of the server. The client creates a socket bound to the hostname and server port 7896. It makes a *DataInputStream* and a *DataOutputStream* from the socket's input and output streams, then writes the message to its output stream and waits to read a reply from its input stream.

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 Sec 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){/*close failed*/}}
    }
}
```

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());}
    }
}
```

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

Java API for TCP streams (cont.)

- The server program opens a server socket on its server port (7896) and listens for *connect* requests. When one arrives, it makes a new thread in which to communicate with the client. The new thread creates a *DataInputStream* and a *DataOutputStream* from its socket's input and output streams and then waits to read a message and write the same one back.
- As our message consists of a string, the client and server processes use the method *writeUTF* of *DataOutputStream* to write it to the output stream and the method *readUTF* of *DataInputStream* to read it from the input stream.
- When a process has closed its socket, it will no longer be able to use its input and output streams. The process to which it has sent data can read the data in its queue, but any further reads after the queue is empty will result in an *EOFException*. Attempts to use a closed socket or to write to a broken stream result in an *IOException.*

External Data Representation and Marshalling

- The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes.
- Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival.
- The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures.

External Data Representation and Marshalling (cont.)

- There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last.
- Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character.

External Data Representation and Marshalling (cont.)

- One of the following methods can be used to enable any two computers to exchange binary data values:
  - The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.
  - The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

External Data Representation and Marshalling (cont.)

- Bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*.
- ***Marshalling*** is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus marshalling consists of the translation of structured data items and primitive values into an external data representation. Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

External Data Representation and Marshalling (cont.)

Three **alternative approaches** to external data representation and marshalling are:

1. CORBA's common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages.
2. Java's object serialization, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

External Data Representation and Marshalling- alternative approaches (cont.)

3. XML (Extensible Markup Language), which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

External Data Representation and Marshalling- alternative approaches (cont.)

- In the first two cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures.

External Data Representation and Marshalling- alternative approaches (cont.)

- In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation. The HTTP protocol is another example of the textual approach.
- Another issue with regard to the design of marshalling methods is whether the marshalled data should include information concerning the type of its contents. For example, CORBA's representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called *namespaces*.

External Data Representation and Marshalling- alternative approaches (cont.)

- Although we are interested in the use of an external data representation for the arguments and results of RMIs and RPCs, it does have a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.

Extensible Markup Language (XML)

- XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term *markup language* refers to a textual encoding that represents both a text and details as to its structure or its appearance. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language. HTML (see Section 1.6) was designed for defining the appearance of web pages. XML was designed for writing structured documents for the Web.
- XML data items are tagged with ‘markup’ strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. That is, in XML, the tags relate to the structure of the text that they enclose, in contrast to HTML, in which the tags specify how a browser could display the text.

Extensible Markup Language (XML)

- XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services. However, XML is also used in many other ways, including in archiving and retrieval systems – although an XML archive may be larger than a binary one, it has the advantage of being readable on any computer. Other examples of uses of XML include for the specification of user interfaces and the encoding of configuration files in operating systems.
- XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them. For example, clients usually use SOAP messages to communicate with web services. SOAP is an XML format whose tags are published for use by web services and their clients.

Extensible Markup Language (XML)

- Some external data representations (such as CORBA CDR) do not need to be self describing, because it is assumed that the client and server exchanging a message have prior knowledge of the order and the types of the information it contains. However, XML was intended to be used by multiple applications for different purposes. The provision of tags, together with the use of namespaces to define the meaning of the tags, has made this possible. In addition, the use of tags enables applications to select just those parts of a document it needs to process: it will not be affected by the addition of information relevant to other applications.

Extensible Markup Language (XML)

- XML documents, being textual, can be read by humans. In practice, most XML documents are generated and read by XML processing software, but the ability to read XML can be useful when things go wrong. In addition, the use of text makes XML independent of any particular platform. The use of a textual rather than a binary representation, together with the use of tags, makes the messages large, so they require longer processing and transmission times, as well as more space to store.
- However, files and messages can be compressed – HTTP version 1.1 allows data to be compressed, which saves bandwidth during transmission.

XML elements and attributes

- Figure shows the XML definition of the *Person* structure that was used to illustrate marshalling in CORBA CDR and Java.

XML definition of the *Person* structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

- It shows that XML consists of tags and character data. The character data, for example *Smith* or *1984*, is the actual data. As in HTML, the structure of an XML document is defined by pairs of tags enclosed in angle brackets.
- In figure, <*name*> and <*place*> are both tags. As in HTML, layout can generally be used to improve readability. Comments in XML are denoted in the same way as those in HTML.

Elements

- An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in previous figure consists of the data *Smith* contained within the <*name*> ... </*name*> tag pair. Note that the element with the <*name*> tag is enclosed in the element with the <*person id="123456789"*> ...</*person* > tag pair. The ability of an element to enclose another element allows hierarchic data to be represented – a very important aspect of XML. An empty tag has no content and is terminated with /> instead of >. For example, the empty tag <*european*/> could be included within the <*person*> ...</*person*> tag.

Attributes

- A start tag may optionally include pairs of associated attribute names and values such as *id="123456789",* as shown above. The syntax is the same as for HTML, in which an attribute name is followed by an equal sign and an attribute value in quotes. Multiple attribute values are separated by spaces.
- It is a matter of choice as to which items are represented as elements and which ones as attributes. An element is generally a container for data, whereas an attribute is used for labelling that data. In our example, *123456789* might be an identifier used by the application, whereas *name*, *place* and *year* might be displayed. Also, if data contains substructures or several lines, it must be defined as an element. Attributes are for simple values.

Names:

- The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon. The names continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case-sensitive. Names that start with *xml* are reserved.

Binary data:

- All of the information in XML elements must be expressed as character data. But the question is: how do we represent encrypted elements or secure hashes The answer is that they can be represented in *base64* notation [Freed and Borenstein 1996], which uses only the alphanumeric characters together with +, / and =, which has a special meaning.

Parsing and well-formed documents

- An XML document must be well formed – that is, it must conform to rules about its structure. A basic rule is that every start tag has a matching end tag. Another basic rule is that all tags are correctly nested – for example, <*x*>..<*y*>..</*y*>..</*x*> is correct, whereas <*x*>..<*y*>..</*x*>..</*y*> is not. Finally, every XML document must have a single root element that encloses all the other elements. These rules make it very simple to implement parsers for XML documents. When a parser reads an XML document that is not well formed, it will report a fatal error.

Remote object references

- When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Remote object references are also passed as arguments and returned as results of remote method invocations, that each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. Here, we discuss the external representation of remote object references.

Remote object references (cont.)

- Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Even after the remote object associated with a given remote object reference is deleted, it is important that the remote object reference is not reused, because its potential invokers may retain obsolete remote object references. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.
- There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of its host computer and the port number of the process that created it with the time of its creation and a local object number. The local object number is incremented each time an object is created in that process.

Remote object references (cont.)

- The port number and time together produce a unique process identifier on that computer. With this approach, remote object references might be represented with a format such as that shown in Figure.

Representation of a remote object reference

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

- In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as the address of the remote object. In other words, invocation messages are sent to the Internet address in the remote reference and to the process on that computer using the given port number. To allow remote objects to be relocated into a different process on a different computer, the remote object reference should not be used as the address of the remote object.

Multicast communication

- The pairwise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary, for example, when a service is implemented as a number of different processes in different computers, perhaps to provide fault tolerance or to enhance availability.
- A *multicast operation* is more appropriate – this is an operation that 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 multicast protocol provides no guarantees about message delivery or ordering.

Multicast communication (cont.)

- Multicast messages provide 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 servers. 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.
2. ***Discovering services 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.

Multicast communication (cont.)

3. ***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.
4. ***Propagation of event notifications*:** Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish subscribe protocols may make use of group multicast to disseminate events to subscribers

IP multicast – An implementation of multicast communication

- **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 a Class D Internet address- – that is, an address whose first 4 bits are 1110 in IPv4.
- 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 (cont.)

- 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. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group. 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. The following details are specific to IPv4:

IP multicast (cont.)

The following details are specific to IPv4:

- ***Multicast routers*:**
  - IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagrams to routers on other networks, where they are again multicast to local members.
  - 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.

IP multicast (cont.)

- ***Multicast address allocation*:** Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). The management of this address space is reviewed annually, with current practice documented in RPC 3171 [Albanna *et al.* 2001]. This document defines a partitioning of this address space into a number of blocks, including:
- Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
- Internet Control Block (224.0.1.0 to 224.0.1.225).
- Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
- Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

IP multicast (cont.)

- Multicast addresses may be permanent or temporary. Permanent groups exist even when there are no members – their addresses are assigned by IANA and span the various blocks mentioned above. For example, 224.0.1.1 in the Internet block is reserved for the Network Time Protocol (NTP), and the range 224.0.6.000 to 224.0.6.127 in the ad hoc block is reserved for the ISIS project. Addresses are reserved for a variety of purposes, from specific Internet protocols to given organizations that make heavy use of multicast traffic, including multimedia broadcasters and financial institutions.
- The remainder of the multicast addresses are available for use by temporary groups, which must be created before use and cease to exist when all the members have left. When a temporary group is created, it requires a free multicast address to avoid accidental participation in an existing group. The IP multicast protocol does not directly address this issue. If used locally, relatively simple solutions are possible – for example setting the TTL to a small value, making collisions with other groups unlikely.
- A client-server solution is adopted whereby clients request a multicast address from a multicast address allocation server (MAAS), which must then communicate across domains to ensure allocations are unique for the given lifetime and scope.

Failure model for multicast datagrams

- Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. That is, some but not all of the members of the group may receive it. This can be called *unreliable* multicast, because it does not guarantee that a message will be delivered to any member of a group.

Java API to IP multicast

- 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. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a specified local port (6789, in next Figure), or any free local port.
- A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket. Effectively, 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. A process can leave a specified group 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);
          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();}
    }
}
```

Java API to IP multicast (cont.)

- In the example, the arguments to the *main* method specify a message to be multicast and the multicast address of a group (for example, *"228.5.6.7").* After joining that multicast group, the process makes an instance of *DatagramPacket* containing the message and sends it through its multicast socket to the multicast group address at port 6789. After that, it attempts to receive three multicast messages from its peers via its socket, which also belongs to the group on the same port. 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 those that joined after it.

Java API to IP multicast (cont.)

- 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.
- An application implemented over IP multicast may use more than one port. For example, the MultiTalk application, which allows groups of users to hold text based conversations, has one port for sending and receiving data and another for exchanging control data.

Reliability and ordering of multicast

- A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message. Also, when a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.
- Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so.
- Ordering is another issue. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members. In addition, messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

Some examples of the effects of reliability and ordering

***1. Fault tolerance based on replicated services*:** Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another.

- This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.

Some examples of the effects of reliability and ordering (cont.)

***2. Discovering services in spontaneous networking*:** One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services. In fact, Jini uses IP multicast in its protocol for discovering services.

***3. Better performance through replicated data*:** Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.

***4. Propagation of event notifications*:** The particular application determines the qualities required of multicast. For example, the Jini lookup services use IP multicast to announce their existence.

Some examples of the effects of reliability and ordering (cont.)

- These examples suggest that some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them.
- The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.