

## Contents of lecture 5:

### 1. TCP – Stream Sockets

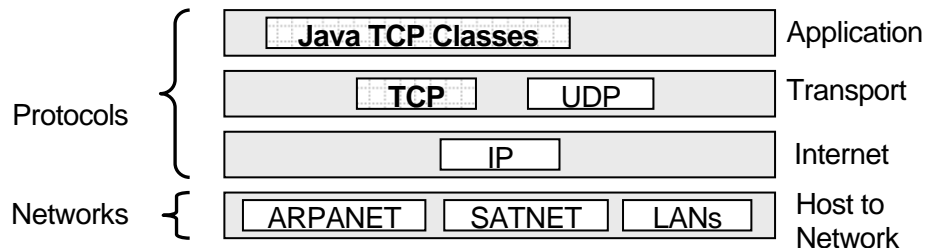
- TCP Protocol/Packet header
- Stream Socket classes in Java

### 2. TCP Programming Examples

- Examples of TCP programming using Java

**Reading Material: UDP Multiuser Chat Program**

The figure opposite places the content of this lecture in relation to the wider TCP/IP model:



# Transmission Control Protocol

## Transmission Control Protocol (TCP)

The goal of TCP is to provide a reliable end-to-end byte stream over an unreliable network. This is problematic as networks tend to have varying topologies, bandwidths, delays, packet sizes, etc. TCP was designed to adapt to the properties of the network, and to be robust in failure. TPC is defined in the international document RFC 1122, with later extensions specified within RFC 1323.

TCP is constructed to accept a stream of data from an application, break it up into smaller pieces not exceeding 64K bytes in size (a size of 1,500 bytes is typical), and pass each packet of data to the IP layer to be sent as an IP datagram.

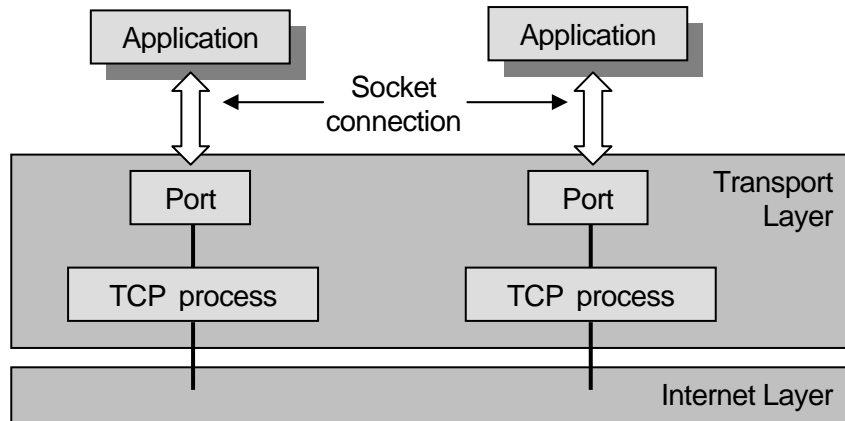
When the IP datagrams arrive at the destination machine, they are passed to the TCP process which then reconstructs the original byte stream (ensuring that the data is concatenated in the correct order).



## TCP and sockets

### Sockets

In order to communicate, both the sender and receiver must create end points called **sockets** across which all data flows. Each socket has an address, consisting of the IP address of the host and the **port** to which it is connected, i.e.



There are two different types of sockets:

- **Active socket**, which is connected via an open data connection (i.e. data can be transferred using the connection). Closing the data connection will destroy the socket.
- **Passive socket**, which is not connected (i.e. data cannot be transferred using a passive socket). Instead, a passive socket waits for a connection attempt, and then spawns an active socket.

Hence, one passive socket, and potentially many active sockets can be associated with every port.

### Functionality of TCP sockets

TCP sockets offer the following functionality (which is mirrored within Java):

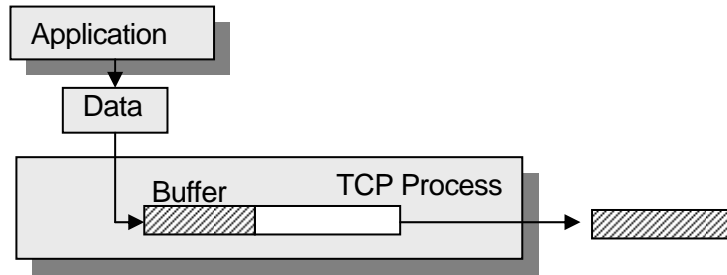
Action	Meaning
SOCKET	Create a new communications socket
BIND	Attach an address to a socket
LISTEN	Announce acceptance of connections
ACCEPT	Wait until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Hence, using TCP sockets, two communication end-points can be established and a data link formed between them, over which data can be passed. Note, a socket connection is full-duplex, in that data can be both sent and received.

### How TCP operates (i.e. TCP Protocol)

Once an active socket has been formed between two applications they are then capable of sending data across or receiving data from the socket.

Whenever an application passes data to a TCP process it may not be immediately sent to the destination host. Instead, the information may be buffered (in order to collect sufficient data to 'fill up' a packet), e.g.:



Applications can specify that data is immediately transmitted through the use of the **PUSH** command. Whenever the TCP process receives a PUSH command it immediately sends the received data (and any buffered contents) onwards.

TCP also supports the notion of **urgent data**, e.g. when an interrupt key is hit. In this case, the TCP process immediately sends onwards its accumulated data with an urgent flag. The receiving TCP process informs the application that the data is urgent.

Information to be sent is ultimately packaged into a TCP segment that consists of a 20-byte header followed by an optional number of data bytes. The segment (header + data) must fit within the 65,535 byte maximum size supported by the IP packet format. Segments that are too long can be broken up into a number of smaller pieces.

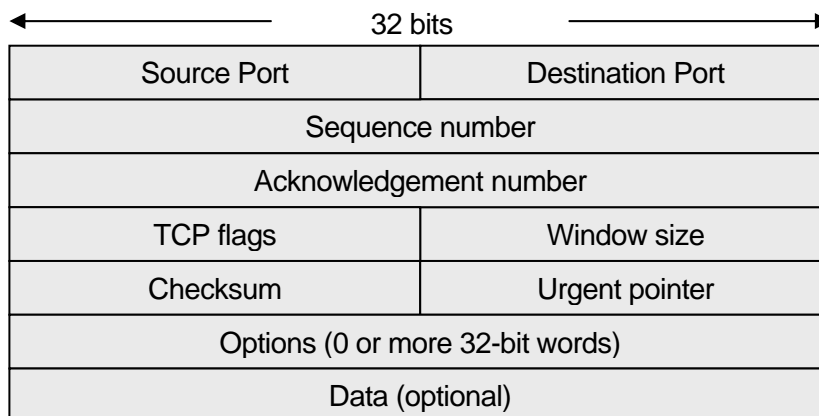
In essence the TCP protocol operates on the principle of delayed transmission. When a sender transmits a segment it also starts a timer. The timer is set to give sufficient time for the segment to arrive with the receiving TCP process and for it to send back an acknowledgment segment (specifying the id of the *next* segment it is expecting to receive). If no acknowledgement is received and the timer runs out, then the packet is sent again.

A number of complex algorithms are employed by TCP processes to ensure the timer approach operates efficiently, dealing with out of sequence packets, lost packets, different routing, etc. However, this is beyond the scope of this particular course.



## TCP Header

The following format is used for the TCP segment header.



Note, segments without any data are commonly used for acknowledgements and control messages.

A breakdown of each of the fields contained within the TCP header follows:

- The source and destination ports define the end points of the connection.
- The sequence number identifies from which part of the entire data stream this segment belongs.
- The acknowledge number is used to acknowledge receipt of the segment. However, within the TCP header, the number specified is actually the number of the segment that is next expected, e.g. if segment 10 has just arrived, then the acknowledgement number is 11.

The TCP flags can be broken up as follows:



Where the *TCP Header Length* specifies the length of the header (as a number of Options words may be contained). Next a 6-bit unused field follows (for future expansion). Then six 1-bit flags; URG – Urgent segment identifier. ACK – identifies that the segment contains a valid acknowledgement number. PSH – pushed data (i.e. do not buffer once received). RST – Reset connection. SYN – used to establish connections. FIN – used to release connections.

- The Window size specifies the maximum number of bytes that can be sent within the segment (used for acknowledgements). For example, zero if the receiver cannot presently process any more data.
- The Checksum is provided as a means of checking if the data was correctly transmitted without an error.
- The Urgent Pointer field points to the section of the segment that contains the urgent data (evidently this is only of use should the urgent data flag be set).
- The Options field is used to specify information not covered within the regular header (for example the maximum segment size that can be supported, etc.).



## TCP support within Java

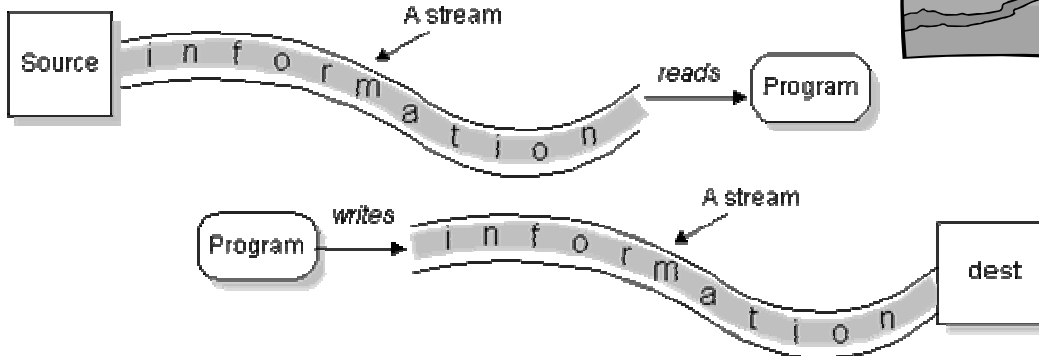
Two principle classes within Java offer a TCP based means of network communication; they are the **Socket** and **ServerSocket** classes. Both classes are now considered. Note, both of these classes draw upon a number of support classes (however, an understanding of the support classes is not needed in order to use TCP sockets).

Before we can explore both the Socket and ServerSocket classes we must firstly have a look at stream processing within Java.

## Streams of Data within Java

A disk based file is one possible source, or sink, for data. However, data may also be gathered from other sources, e.g. a keyboard, network connection, etc. Java employs the notion of a *stream* as a higher level abstraction for all possible data sources.

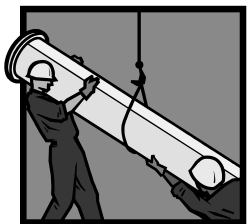
Hence, in Java, a *stream* might be receiving bytes from (or sending them to) a file, a String object, a network socket, the keyboard, etc. Fundamentally, a *stream* is a flow of data, never mind where it comes from or goes to. Functionally, data can be added to, or plucked from a stream.



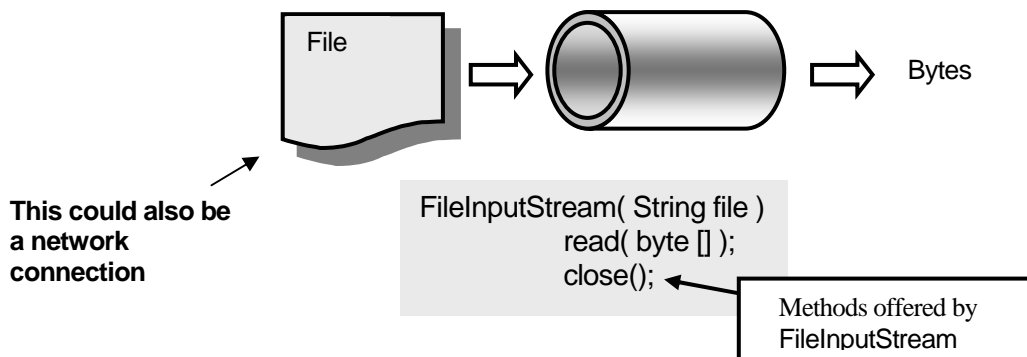
### Layering Streams on top of each other

The following information is not directly related to networking, however, it is pertinent if we are to easily use TCP streams within our Java programs.

Java provides a useful input stream known as `DataInputStream`, which offers methods for reading characters, integers, floats, etc. However, only one constructor is defined for the class: `java.io.DataInputStream( java.io.InputStream )`. Given this, how can a `DataInputStream` be used to read information from a disk file? Within Java, streams are designed to be layered on top of each other, entailing that the programmer must connect together several (usually two) streams to get the exact type of required functionality.



In order to read data from a diskfile we firstly need to use the `FileInputStream`, which possesses the following constructor `FileInputStream( Sting filename )`, i.e. permitting a stream to be opened to a specified disk file.



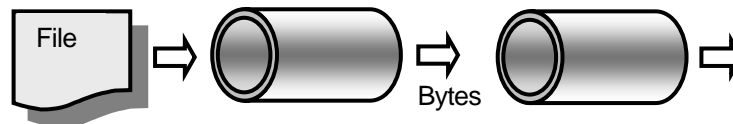
Using a `FileInputStream` a stream connection can be made to a disk file. However, the `FileInputStream` only provides methods for reading bytes from the disk file. In order to read integers, floats, etc. it is necessary to append a `DataInputStream` onto the `FileInputStream`, e.g.

```
FileInputStream fileStream = new FileInputStream( "file.dat" );
DataInputStream dataStream = new DataInputStream( fileStream );
```

or simply just:

```
DataInputStream dataStream =
    new DataInputStream( new FileInputStream( "file.dat" ) );
```

Hence, the `FileInputStream` provides a means of reading an array of bytes from a disk file, and the `DataInputStream` provides a means of converting the bytes into integers, floats, etc.



```
FileInputStream( String file )
read( byte [] );
close();
```

```
DataInputStream(InputStream)
readByte(); readChar();
readBoolean(); readDouble();
```

## Streams and TCP Networking

In what follows a brief overview of the `InputStream/OutputStream` classes is firstly presented, followed by a breakdown of the `DataInputStream/DataOutputStream` classes.

The TCP classes within Java will typically provide the programmer with an `InputStream` or `OutputStream` to be used for the actual communication. As we will see, the functionality of the `InputStream/OutputStream` classes is actually very limited – simply providing the programmer with an ability to read/write bytes. Hence, programmers will normally link a `DataInputStream/DataOutputStream` to the basic `InputStream/OutputStream` so that they can readily read Strings, floats, integers, etc.



### The `InputStream` and `OutputStream` Classes

Principally these classes offer a means of reading or writing bytes of information (via the `read( byte b[] )` and `write( byte b[] )` methods). Both classes also provide a `close()` method to terminate any data transfer. The `OutputStream` class also offers a `flush()` method to ensure that any buffered output bytes are written.



### The `DataInputStream` and `DataOutputStream` Classes

The `DataInput/OutputStream` classes extend the functionality of the `Input/OutputStream` classes. As before functions are included for sending arrays of bytes, but now, methods are provided for sending characters (individually or as an array), numbers (integers, floating point, short ints), etc.

The `writeUTF( String str )` method is deserved of a special mention. The method is intended to write out an unformatted text field. This method is often used when sending Strings.

An overview of useful methods within these classes now follows:

```
public class java.io.DataOutputStream
{
    public DataOutputStream( OutputStream out );

    public void flush();
    public void write( byte b[], int off, int len);
    public final void writeBoolean( boolean v );
    public final void writeByte( int v );
    public final void writeBytes( String s );
    public final void writeChar( int v );
    public final void writeChars( String s );
    public final void writeDouble( double v );
    public final void writeFloat( float v );
    public final void writeInt( int v );
    public final void writeLong( long v );
    public final void writeShort( int v );
    public final void writeUTF( String str );
}
```

```
public class java.io.DataInputStream
{
    public DataInputStream( InputStream in );

    public final int read( byte b[] );
    public final boolean readBoolean();
    public final byte readByte();
    public final char readChar();
    public final double readDouble();
    public final float readFloat();
    public final int readInt();
    public final String readLine();
    public final long readLong();
    public final short readShort();
    public final String readUTF();
    public final int skipBytes( int n );
}
```

Given the above, we can now more fully explore both the `Socket` and `ServerSocket` classes.

## Socket class

The **Socket** class implements an active socket (i.e. a TCP end-point that can be used to transfer information). The class is as follows:

### public class Socket

└ extends java.lang.Object

**Overview:** A **Socket** object is simply used as a communications end-point. They are initiated from the client process (i.e. the **ServerSocket** class deals with server side connections). An overview of the most common constructors and methods within the class follows:

Constructor Summary	
public Socket( InetAddress address, int port )	Creates a socket and binds it to the specified IP/port number
public Socket( InetAddress address, int port, InetAddress localAddr, int localport )	Creates a socket and binds it to the specified end points
public Socket( String host, int port )	Creates a socket and binds it to the specified host/port
public Socket( String host, int port, InetAddress localAddr, int localport )	Creates a socket and binds it to the end points.

A breakdown of the constructors within this class follows:

```
public Socket( InetAddress address, int port ) throws IOException
```

This method creates a **Socket** object and connects it to the specified port number at the specified IP address. If there is a security manager, then the `checkConnection` method is called to determine if this operation is permitted, throwing a **SecurityException** if not permitted.

This is probably the most commonly used constructor, providing a means of connecting to a specific IP/port, using the first available port on the local host at the start point.

```
public Socket( InetAddress address, int port, InetAddress localAddr, int localPort )  
               throws IOException
```

This method creates a socket object and binds it to the specified remote address/port and also the specified local address/port. As before, the security manager is consulted to see if this operation is permitted. This particular constructor should be used if the start point needs to be bound to a particular port/IP on the local machine.

```
public Socket( String host, int port ) throws IOException, UnknownHostException
```

This method is similar to the `Socket(InetAddress, int)` method, except that a `String` host is specified. An `UnknownHostException` is thrown if the host cannot be found.

```
public Socket( String host, int port, InetAddress localAddr, int localPort )  
               throws IOException, UnknownHostException
```

This method is similar to the `Socket(InetAddress, int, InetAddress, int)` method except that the destination end-point is specified in a `String` format.

A summary of the most commonly used methods now follows:

Method Summary	
close()	Closes the socket
getInetAddress()	Returns the InetAddress to which the socket is connected
getPort()	Returns the port to which the socket is connected
getLocalAddress()	Gets the local address to which the socket is connected
getLocalPort()	Returns the local port to which the socket is connected
getOutputStream()	Gets an output stream for the socket
getInputStream()	Returns an input stream for the socket

Aside: The **Socket** class also contains a number of other methods for setting various TCP flags (e.g. `urgent`, `do_no_buffer`) and controlling the buffer size.

#### public void **close()** throws IOException

This method simply closes an open socket connection. If an I/O error occurs when attempting to close the socket then an **IOException** is thrown.

Any opened sockets should be closed before the application quits to ensure the relevant port can be successfully reused.

#### public InetAddress **getInetAddress()**

Returns the remote IP address to which the socket is connected. This method is useful if a connection was made using a **String** formatted host, or a number of socket connections are maintained within the program.

#### public int **getPort()**

Returns the remote port to which the socket is connected.

#### public InetAddress **getLocalAddress()**

This method returns the local socket address.

#### public int **getLocalPort()**

This method returns the local port to which the socket is connected. When a socket connection is made to the first available port, this method can be used to determine the port that was used.

#### public OutputStream **getOutputStream()** throws IOException

This method returns an output stream for the socket. Using the output stream an application can send data across the socket to the destination host. An **IOException** will be thrown should any I/O errors arise.

**public InputStream getInputStream() throws IOException**

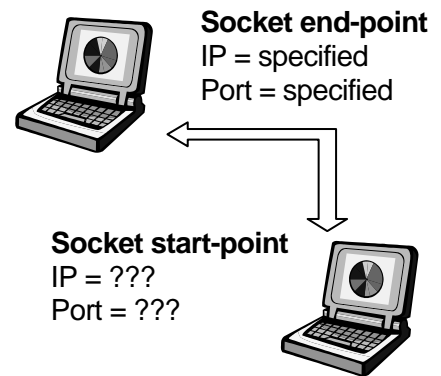
This method returns an input stream for the socket. Using the input stream an application can receive information via the socket from the destination host. An **IOException** will be thrown should any I/O errors arise.

To recap, the **Socket** object is primarily of use to establish connections between different hosts. A client process will create a **Socket** object and attempt to connect it to a specified IP/port number. Once a connection has been established, data communication can occur via the **getOutputStream** and **getInputStream** methods.



When creating a **Socket** object it is necessary to explicitly specify the IP/port to which it should connect. This is not necessarily true of the local host. For example, the **Socket( IP, port )** method only requires that the end point is specified, with the first free available port used as the start point.

As such, **Socket( IP, port )** is best interpreted as meaning 'Create a socket to this specified end-point, and use any free port on the local machine as the socket start-point', i.e. it does not matter which start-point was used. If needed, the socket start-point can be retrieved through the **getLocalPort** and **getLocalAddress** methods.



**ServerSocket class**

The **ServerSocket** class acts as a server object for TCP **Socket** connection requests. In particular, this class listens on a particular port for any incoming socket connection requests, and returns a **Socket** object that the server can use to communicate with any host that connects to the server. The class is as follows:

**public class ServerSocket**

└ extends java.lang.Object

**Overview:** The **ServerSocket** class offers methods that permit a particular port to be monitored, and any incoming socket formation requests to be processed. An overview of the most commonly used constructors and methods follows:

Constructor Summary	
public ServerSocket( int port )	Creates a server socket on a specified port
public ServerSocket( int port, int backlog )	Creates a server socket on the specified port with backlog queue length

A breakdown of the constructors follows:

**public ServerSocket( int port ) throws IOException**

This method creates a server socket on a specified port. If a port number of 0 is specified, then a socket server is created on the first free port found ( > 1024 ). A default queue length of 50 connections is assumed (i.e. 50 incoming connection requests, can be queued awaiting processing, before additional connection requests are refused).

Note, this method checks with the security manager (should one be in force) to ensure that the operation is permitted (throwing a **SecurityException** if this is not the case). Any I/O problems will result in an **IOException** being generated.

```
public ServerSocket( int port, int backlog ) throws IOException
```

This method is the same as the **Socket**( int port ) constructor, except that the number of queued connection requests is also specified. A summary of the most commonly used methods now follows:

Method Summary	
<code>accept()</code>	Listens for a connection to this socket and accepts it
<code>close()</code>	Closes the server socket
<code>getInetAddress()</code>	Returns the local IP of the server socket
<code>getLocalPort()</code>	Returns the port on which the socket is listening
<code>toString()</code>	Returns the local IP address and port as a String object

As with the **Socket** class, some methods have been omitted.

```
public Socket accept() throws IOException
```

This method waits for a connection to be made to this socket. Once a connection attempt is received, the **ServerSocket** object will accept the connection. The **ServerSocket** object 'blocks' until a connection is made (i.e. whatever thread it is running within, inside the server, is put on hold – or, in other words, whenever **accept** is called the program will stop and wait until an incoming connection is received).



Once a connection is made, a new **Socket** object is spawned through which communication can proceed (Note, the security manager is firstly checked to see if this operation is permitted). Should any I/O communication problems arise then an **IOException** is thrown.

Aside: A **ServerSocket** object can be considered as an instance of a passive socket (as defined earlier), and a **Socket** object can be considered an instance of an active socket.

```
public void close() throws IOException
```

This method closes the server socket, issuing an **IOException** should any problems arise.

```
public InetAddress getInetAddress()
```

Returns the IP address to which the socket is connected, or alternatively **null** if the server socket is not yet connected.

```
public int getLocalPort()
```

Returns the port on which the server socket is listening.

```
public String toString()
```

Returns the local server IP address and port formatted as a **String**.

## Client-server interaction using Sockets

Normally a server runs on a specific computer, listening to a particular port (using a **ServerSocket**) and waiting for a client to make a connection request.

On the client-side, the client must know the hostname/IP of the server, and the port on which the server is running. Given this information the client can attempt a connection (**Socket** class).

If all goes well, then the server will accept the incoming request. Upon connection, the server generates a new **Socket** object (bound to a different port, i.e. different than the port used to listen for incoming connection requests), thereby permitting the server to continue listening for additional connection requests whilst tending to the needs of the connected client.

Client-side, once the connection is accepted, the socket can then be used to communicate with the server (across whatever port the client's socket was bound to). The process can be summarised as follows:

### The Sending Computer

Let us assume we have some data we want to send to another computer. In order to do this:

- The sending computer must know the IP address and port number on which the destination **ServerSocket** is listening.
- Create a **Socket** object and bind it to the IP/port that contains the listening **ServerSocket** object.
- If the connection is successful, then the **getInputStream** and **getOutputStream** methods of the **Socket** object can be called to enable communication to occur between the connected hosts.

### The Receiving Computer

In order to receive a datagram we must:

- Firstly create a **ServerSocket** object and bind it to the port on which we expect incoming connections to be made.
- Call the **accept** method of the **ServerSocket** object and wait for an incoming connection.
- Once a connection has been received, the **accept** method will return a **Socket** object that can be used to communicate with the connecting computer.
- If the connection is successful, then the **getInputStream** and **getOutputStream** methods of the **Socket** object can be called to enable communication to occur between the connected hosts.

Using socket programming, it is possible to transfer data, in a full-duplex manner, between the client and server. However, this is only half of the story. It is the responsibility of the programmer to ensure the client and server know how to successfully communicate with one another (i.e. development of a protocol for the application).

The governing protocol will determine what happens after the connection is made. In order for the client and server to communicate, they must both implement some mutually acceptable application protocol.



## Sample Program 1: Date/time program

This example program makes use of the *daytime* port on a computer (which if correctly configured will return the local data and time on that computer). The *daytime* port is number 13

As before, we need some code to deal with exceptions:

```
import java.net.*; import java.io.*; import java.util.*;
public class TimeDate
{
    public static void main(String[] args)
    {
        String server = "www.whitehouse.net";
        try
        {
            // Get an input stream connection
            Socket socket = new Socket( server, 13 );
            InputStream inputStream = socket.getInputStream();

            DataInputStream dataInputStream
                = new DataInputStream( inputStream );

            // Print out the time at the server
            System.out.println( "\nTime at: "+server+":" );
            System.out.println( dataInputStream.readUTF() );

            // Print out the current time
            System.out.println( "\nCurrent time: " );
            System.out.println( new Date() );

            // Clean up
            inputStream.close(); socket.close();
        }
        catch(UnknownHostException e)
        {
            System.out.println(e);
            System.out.println( "Cannot connect to host." );
        }
        catch( IOException e )
        {
            System.out.println( e );
            System.out.println( "\nCommunication probs" );
        }
    }
}
```

The above program simply creates a socket connection to the desired host/port, and from that obtains an input stream from which the date/time can be received (whenever a socket connects to the *daytime* port, the date/time is automatically sent by the server, i.e. this is the protocol adopted by the *daytime* port).

When run, the following output is obtained:

```
Time at: www.whitehouse.net:
Wed Nov 3 07:10:10 2002

Current time:
Wed Nov 03 12:12:46 GMT 2002
```

A particular port can have separate TCP and UDP connections (i.e. potentially 2 applications can use one port at a time). Often, a particular server will use both TCP/UDP connections on a particular port to support different connections from clients (e.g. echo).

## Sample Problem 2:

The following program is a bit more complex, offering a client-server setup which mirrors the previous DatagramSocket/Packet example provided in the previous lecture. To recap:

- **Server side:** The server simply waits for an incoming connection. Once a client connects the server will wait until a **String** of text is sent over. The server calculates the number of characters received and returns this information to the client. Should the server receive a “quit” **String**, then it terminates the connection with the client (it is important to provide a means of closing the connection).
- **Client side:** The client firstly accepts input from the user, permitting several lines of text to be entered. When the *send* button is clicked by the user, the client will attempt to connect to the server. Once connected, it will send over each line of text entered by the user, and wait for the server’s response. Once all the user’s input has been processed, the client will send over a “quit” **String**, informing the server the conversation is complete.

**Note**, the design outlined above is not ideal – however, it is straightforward. It would be worthwhile asking yourself how you would go about designing an improved approach.



The code now follows:

The code opposite creates the server’s GUI as well as starting the server running. The `runServer` method is defined next.

```
import java.io.*; import java.net.*; import java.awt.*;
import java.awt.event.*; import javax.swing.*;
public class Server extends JFrame
{
    private JTextArea display;

    public Server()
    {
        addWindowListener(
            new WindowAdapter()
            { public void windowClosing( WindowEvent e )
              { System.exit( 0 ); }
            } );

        display = new JTextArea();
        Container contents = getContentPane();
        contents.add( new JScrollPane( display ) );
        setSize( 300, 400 ); show();
    }

    public void runServer() { ...defined below... }

    public static void main( String args[] )
    {
        Server serverApp = new Server();
        serverApp.runServer();
    }
}
```

In particular, a **ServerSocket** object is created, which waits for a connection. Once a connection is received, information on the connecting client is printed. Next, data input and output streams are created, so that information can be exchanged with the client.

```
public void runServer()
{
    try {
        // Create a server socket
        ServerSocket server = new ServerSocket( 10000, 10 );

        // Process server interactions until program exits
        while( true )
        {
            // Wait for a connection
            display.append( "\nWaiting for a connection." );
            Socket connection = server.accept();

            // Display information on the connection
            display.append( "\nConnection accepted: " +
                connection.getInetAddress().getHostName() );

            // Get input and output streams
            DataInputStream input = new DataInputStream(
                connection.getInputStream() );
            DataOutputStream output = new DataOutputStream(
                connection.getOutputStream() );

            // Process interaction from the client
            String message = input.readUTF();
            while( !message.equals( "quit" ) )
            {
                // Display client message and reply
                display.append( "\nClient: " + message );

                output.writeUTF( "Received:" + message +
                    " : Size = " + message.length() );

                message = input.readUTF();
            }

            // Termination connection once "quit" received
            // and wait for another connection.
            input.close();
            output.close();
            connection.close();
        }
    }
    catch( IOException error )
    { error.printStackTrace(); }
}
```

The `runServer` class processes input from the client, waiting for a `String`, before sending back the length of the string. Once a "quit" `String` has been received, the application terminates its connection with the client, and waits for another incoming connection.

The client code now follows:

The first half of the client's constructor sets up the GUI and interactions. The second half of the client contains the `main` and `sendData` methods.

The client has been structured, so that the user can add lines of text to a **Vector** by using the **JTextField** object. Whenever, the user hits the **JButton**, the **sendData** method is called, which attempts to connect to the server, and then exchange data.

```
import java.io.*; import java.net.*; import java.awt.*;
import java.awt.event.*; import javax.swing.*; import java.util.*;

public class Client extends JFrame
{
    private JTextArea display; private JTextField enter;
    private JButton startButton; private Vector inputText;

    public Client()
    {
        addWindowListener( new WindowAdapter()
        {
            public void windowClosing( WindowEvent e )
            { System.exit( 0 ); }
        });

        inputText = new Vector();
        display = new JTextArea();
        display.setText( "Enter some text, then hit button." );

        // Define a text field for entering text
        enter = new JTextField();
        enter.addActionListener( new ActionListener()
        {
            public void actionPerformed( ActionEvent event )
            {
                inputText.addElement( enter.getText() );
                display.append( "\nLine " + inputText.size()
                + " : " + enter.getText() );
                enter.setText( "" );
            }
        });

        // Define a button for sending data
        startButton = new JButton( "Send Data" );
        startButton.addActionListener( new ActionListener()
        {
            public void actionPerformed( ActionEvent event )
            { sendData( inputText ); }
        });

        Container contents = getContentPane();
        contents.add( enter, BorderLayout.NORTH );
        contents.add( new JScrollPane( display ),
            BorderLayout.CENTER );
        contents.add( startButton, BorderLayout.SOUTH );

        setSize( 300, 400 ); show();

    } // End of client's constructor

    // Send data to the server
    private void sendData( Vector inputText ) { // Define later }

    public static void main( String args[] )
    {
        Client clientApp = new Client();
    }
}
```

The `sendData` method is now defined.

The method attempts to connect to the server. Once connected each line of entered text is transmitted (a reply is received before the next line is sent).

When all the text has been processed the “quit” message is sent (Note, evidently this is a dangerous design as the user might enter ‘quit’ as a string to sent to the server, thereby prematurely severing the connection).

```
private void sendData( Vector inputText )
{
    try {        // Try to connect to the server
        display.append( "\nAttempting to connect to server" );
        Socket connection = new Socket(
            InetAddress.getLocalHost(), 10000 );

        // Create input and output streams
        display.append( "\nCreating input and output streams." );
        DataInputStream input = new DataInputStream(
            connection.getInputStream() );
        DataOutputStream output = new DataOutputStream(
            connection.getOutputStream() );

        // Send over data entered by user
        for( int idx = 0; idx < inputText.size(); idx++ )
        {
            display.append( "\nSending line " + (idx+1) );
            output.writeUTF( (String)inputText.elementAt( idx ) );
            String response = input.readUTF();
            display.append( "\nResponse: " + response );
        }

        // Inform the server the communication is finished
        output.writeUTF( "quit" );

        // Close down connections
        input.close(); output.close(); connection.close();
    }
    catch( IOException error )
    { error.printStackTrace(); }
}
```

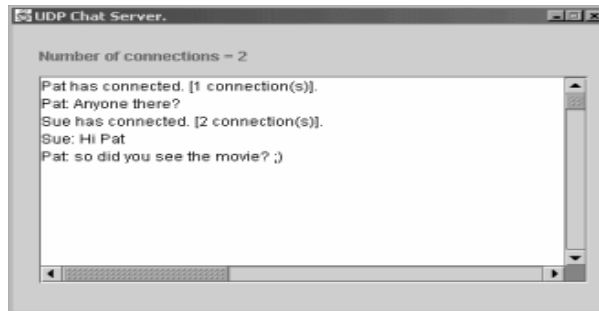
Aside: Woes of running network applications – You should be aware that if a network based application crashes, without first freeing the port it was using for communication, then it may not be possible to use that port again (as the computer still regards the crashed program as controlling the port). In this case, either use another free port, or reboot the computer.



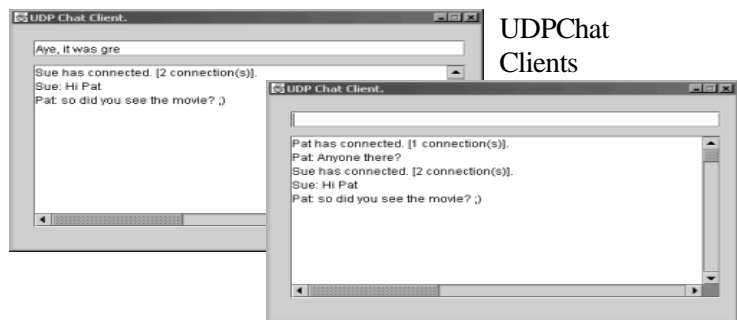
## Optional: UDP Based, multi-user, chat program

In what follows a sizeable, multi-user, UDP chat program will be outlined. The program is simply provided as an aside: something you should study with an aim of taking away anything useful (the level of programming complexity is beyond that needed to pass the module, however, having said this, it is certainly within the grasp of most students).

Graphically the UDP Chat program is as shown:



UDPChat  
Server



UDPChat  
Clients

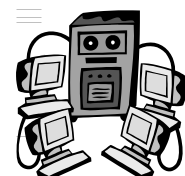
### Client Side Functionality

- When the client is first run, it should prompt the user for the IP of the server to connect to. After this, it should then request the user to give their nickname. Once both pieces of information have been obtained, the client can then connect to the server (by sending a packet containing a connection code and the nickname of the user).
- Graphically, the client should contain a text area onto which any messages sent from the server are displayed. The client should also make use of a textfield, into which the user may enter messages to be sent to the server, i.e. once the user hits return, the contents of the textfield are extracted and sent to the server.
- Any messages sent back from the server should simply be displayed in the text area.



### Server Side Functionality

- When run, the server should simply wait for any packets received from clients. Received packages can contain either a connection request or alternatively a message to be sent to all connected clients.
- If the received packet contains a connection request, then the server should add the client's IP and port number to its list of connected clients (assuming there is still space to do so). Additionally, a message should then be broadcast to all clients, informing them of the new connection (and showing the nickname of the connected client).
- If the received packet contains data (i.e. a text message), then it should be sent to all connected clients.
- The server should also maintain a text area that should echo any connection requests, or messages sent by users.



The first part of the client defines variables used within the program, including GUI components, communication specific items, UDP connection variables, etc.

```
// Imports – lots of them
import java.net.*; import javax.swing.*;
import java.awt.event.*; import javax.swing.border.*;
import java.awt.*; import java.io.*; import java.util.*;

public class UDPChatClient extends JFrame
{
    // Define GUI components
    private JTextField message;           // Message to send
    private JTextArea textMessages;      // Received messages

    // Chat variables
    public String nickName;              // Nickname of user

    // Define variables for communication
    private boolean bConnected = false;
    public String strConnectionIP;
    private int iServerPortNum = 10000;

    // Define UDP variables
    private DatagramSocket socket;
    private DatagramPacket sendPacket;
    private byte[] sendData;
    private DatagramPacket receivePacket;
    private byte[] receiveData;
}
```

The next part of the client provides both the constructor and the main method. The constructor basically creates the GUI and also asks which IP the server is running on and the chat nickname that should be used. The main kick starts the application, including a call to the connection manager.

```
// Constructors
public UDPChatClient()
{
    // Give our program a title.
    super( "UDP Chat Client." );

    // Create and display the GUI
    setupGUI();

    // Obtain the server connection IP
    // This will update strConnectionIP
    GetIPDialog ipDialog = new GetIPDialog( this );

    // Obtain the nick name of the user
    // This will update nickName
    GetNickName nicknameDialog
        = new GetNickName( this );
}

// Start the program running
public static void main( String args[] )
{
    UDPChatClient instance = new UDPChatClient();
    instance.startConnectionManager();
}
```

The first part of the client's `setupGUI` follows (called from within the constructor). Apart from sizing the application and providing a means of quitting, code is also provided that creates the text area into which received messages are displayed.

**// The setupGUI method creates and defines the GUI**

```
private void setupGUI()
{
    // Add window listener permitting program exit
    this.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing( WindowEvent e )
            { System.exit( 0 ); }
        }
    );

    // Use a null layout manager, and size application
    this.getContentPane().setLayout( null );
    this.setSize( 440, 300 );
    this.setResizable( false );

    // Define GUI controls used within the program.
    // Create area to display received messages
    textMessages = new JTextArea( 100, 100 );
    textMessages.setEditable( false );
    JScrollPane textMessagesScrollPane
        = new JScrollPane( textMessages );
    this.getContentPane().add( textMessagesScrollPane );
    textMessagesScrollPane.setBounds( 20, 50, 400, 200 );
    textMessagesScrollPane.setBackground( Color.gray );
}
```

The second part of the `setupGUI` follows, defining the text field into which the user types. Code is provided that extracts the entered text, forms it into a UDP packet, and finally sends it to the server.

```
// Define text field for entering messages
message = new JTextField( 100 );
this.getContentPane().add( message );
message.setBounds( 20, 20, 400, 20 );

// When return is pressed, transmit typed message
message.addActionListener( new ActionListener()
{
    public void actionPerformed( ActionEvent e )
    {
        // Check if connected
        if( bConnected )
        {
            try
            {
                // Construct message, D = data
                String sendMessage = "D" + nickName +
                    ": " + message.getText();
                sendData = sendMessage.getBytes();

                // Construct UDP packet and send
                sendPacket = new DatagramPacket(
                    sendData, sendData.length,
                    InetAddress.getByName( strConnectionIP ),
                    iServerPortNum );
                socket.send( sendPacket );

                // Clear any entered text
                message.setText( "" );
            }
            catch( SocketException error )
            {
                textMessages.setText(
                    "Error: SocketException occurred." );
            }
            catch( IOException error )
            {
                textMessages.setText(
                    "Error: Network IO exception occurred." );
            }
        }
    }
});
}
```

The final part of the `setupGUI` method follows. The code simply provides some `catch` blocks for any network related exceptions (in all cases an error message is displayed and then the program continues).

The `startConnectionManager` now follows. This method is called from within the main method and is responsible for connecting to the server and receiving any messages sent *back* from the server.

```
// Connect to the server, receive incoming messages
public void startConnectionManager()
{
    try
    {
        // Create communications socket.
        socket = new DatagramSocket();

        // Send a connection packet, 'C' = connection.
        sendData = (new String( "C" + nickName )).getBytes();
        sendPacket = new DatagramPacket(
            sendData, sendData.length,
            InetAddress.getByName( strConnectionIP ),
            iServerPortNum );
        socket.send( sendPacket );
        bConnected = true;

        // Continue to receive packets from the server.
        while( true )
        {
            // Wait for an incoming packet
            receiveData = new byte[1000];
            receivePacket = new DatagramPacket( receiveData, 1000 );
            socket.receive( receivePacket );

            // Display the message in the text field
            String message = new String( receiveData, 1,
                receivePacket.getLength()-1 );
            textMessages.append( message + "\n" );
        }
    }
    catch( SocketException e )
    {
        textMessages.setText(
            "Error: SocketException occurred." );
    }
    catch( IOException e )
    {
        textMessages.setText(
            "Error: Network IO exception occurred." );
    }
}
}
```

The event handler attached to the `message` text field handles all messages sent *to* the server.

The first part of the `startConnectionManager` attempts to connect to the server, using the user provided IP. In particular, the client sends both a “C” code – for connection – and the user’s nickname to the server for registration purposes.

The second part of the `startConnectionManager` method continually loops within a `while` block, receiving any packets sent from the server, and displaying the textual content of the packets on the `textMessage` GUI component.

The UDPChatServer code now follows. As with the client, the first slide of the server code defines data variables used within the program. Notice that the server defines storage for up to 100 client connections.

**// UDPChatServer**

```
import java.net.*; import javax.swing.*; import java.io.*;
import java.awt.event.*; import java.awt.*; import java.util.*;

public class UDPChatServer extends JFrame
{
    // Define GUI components
    private JLabel infoLabel;
    private JTextArea textMessages;

    // Define UDP variables
    private DatagramSocket socket;
    private DatagramPacket sendPacket;
    private byte[] sendData;
    private DatagramPacket receivePacket;
    private byte[] receiveData;

    // Define variables for communication
    private int iPortNum = 10000;

    // Define storage for client's connections
    private int iMaxNumClients = 100;
    private int iCurNumClients = 0;
    private InetAddress clientAddress[];
    private int iClientPortNum[];
```

The next part of the server provides the constructor and main methods. The server's constructor simply constructs the GUI, calling setupGUI. The main method starts the application, and calls startConnectionManager.

**// Constructors**

```
public UDPChatServer()
{
    super( "UDP Chat Server." );

    // Create and display the GUI
    setupGUI();
}

// Main method
public static void main( String args[] )
{
    UDPChatServer instance = new UDPChatServer();
    instance.startConnectionManager();
}
```

The next slide provides the `setupGUI` code. The server has a simpler GUI than the client, only needing a text area into which messages sent by the client can be displayed.

```
// setupGUI creates the GUI for this application.
private void setupGUI()
{
    // Add listener permitting exit from program
    this.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing( WindowEvent e )
            { System.exit( 0 ); }
        }
    );

    // Use a null layout manager, size application
    this.getContentPane().setLayout( null );
    this.setSize( 440, 300 );
    this.setResizable( false );

    infoLabel = new JLabel( "Waiting for a connection." );
    this.getContentPane().add( infoLabel );
    infoLabel.setBounds( 20, 20, 400, 20 );

    // Create area for displaying messages
    textMessages = new JTextArea( 100, 100 );
    textMessages.setEditable( false );
    JScrollPane textMessagesScrollPane
        = new JScrollPane( textMessages );
    this.getContentPane().add( textMessagesScrollPane );
    textMessagesScrollPane.setBounds(20,50,400,200 );
    textMessagesScrollPane.setBackground( Color.gray );

    this.setVisible( true );
}
```

The `startConnectionManager` method now follows. The connection manager has the dual responsibility of accepting any incoming connection requests and also sending any text received by a client to all the other connected clients.

The first part of the connection manager simply creates a communications socket and initialises storage space for the connected client's details.

```
// Accept incoming connections / messages
public void startConnectionManager()
{
    try
    {
        // Create client details storage
        clientAddress = new InetAddress[ iMaxNumClients ];
        iClientPortNum = new int[ iMaxNumClients ];

        // Create a datagram socket for connections
        socket = new DatagramSocket( iPortNum );
    }
}
```

The next box shows the while loop defined within the connection manager, which continually loops, accepting incoming datagrams and checking if they are either a connection request or alternatively a text message.

```

while( true )
{
    // Wait for an incoming packet
    receiveData = new byte[1000];
    receivePacket = new DatagramPacket( receiveData, 1000 );
    socket.receive( receivePacket );

    // Determine if connection request or message
    if( receiveData[0] == 'C' )
    {
        // Method defined below
        processConnection(receivePacket, receiveData );
    }
    else if( receiveData[0] == 'D' )
    {
        // Method defined below
        processMessage( receiveData,
            receivePacket.getLength() );
    }
}
}
catch( SocketException e )
{
    infoLabel.setText( "Error: SocketException occurred." );
}
catch( IOException e )
{
    infoLabel.setText( "Network IO exception occurred." );
}
}

```

The processConnection method now follows. This method is called from within startConnectionManager whenever the server receives a connection request datagram. The client is connected if there is sufficient space, and a connection message broadcast to all other users.

```

// processConnection processes new connections
private void processConnection( DatagramPacket,
    receivePacket, byte[] receiveData )
{
    // Check that more clients can connect
    if( iCurNumClients < iMaxNumClients )
    {
        // Add user to list of connected clients
        clientAddress[ iCurNumClients ] = receivePacket.getAddress();
        iClientPortNum[ iCurNumClients ] = receivePacket.getPort();
        iCurNumClients++;
        infoLabel.setText( "Number of connections = "
            + iCurNumClients );

        // Broadcast that the user has joined
        String broadcastMessage = new String(
            receiveData, 0, receivePacket.getLength() )
            + " has connected. [" + iCurNumClients + "
            connection(s)].";
        processMessage( broadcastMessage.getBytes(),
            broadcastMessage.length() );
    }
}
}

```

The final method within the server follows. `processMessage` is used to send a message to all connected clients. In particular, a datagram is sent to each IP defined within the `clientAddress` array.

```
// processMessage is used to broadcast messages
private void processMessage( byte[] broadcastData,
                             int iDataSize )
{
    // Send message to all connected clients
    for( int iIdx = 0; iIdx < iCurNumClients; iIdx++ )
    {
        try
        {
            // Create the packet to send to the client.
            sendPacket = new DatagramPacket(
                broadcastData, iDataSize,
                clientAddress[iIdx], iClientPortNum[iIdx] );

            socket.send( sendPacket );
        }
        catch( IOException e )
        {
            textMessages.append(
                "Error: Network IO exception occurred." );
        }
    }

    // Also display the message on the server
    textMessages.append(
        new String( broadcastData, 1, iDataSize-1 ) + "\n" );
}
}
```

## Practical 5

After this lecture you should explore the fifth practical pack which should enable you to investigate the material in this lecture.

## Learning Outcomes

Once you have explored and reflected upon the material presented within this lecture and the practical pack, you should:

- Have knowledge of the TCP packet format and the functionality defined within TCP sockets. In addition, understand the broad principles behind how TCP operates.
- Have knowledge of the functionality on offer within the `Socket` and `ServerSocket` classes and understand how the functionality facilitates TCP based communication.
- Understand how the `Socket` and `ServerSocket` classes can be used to construct a straightforward client-server relationship and be able to write a Java program that provides such a setup given a straightforward problem description.

More comprehensive details can be found in the CSC734 Learning Outcomes document.

