

Introduction to User Datagram Protocol in Java.

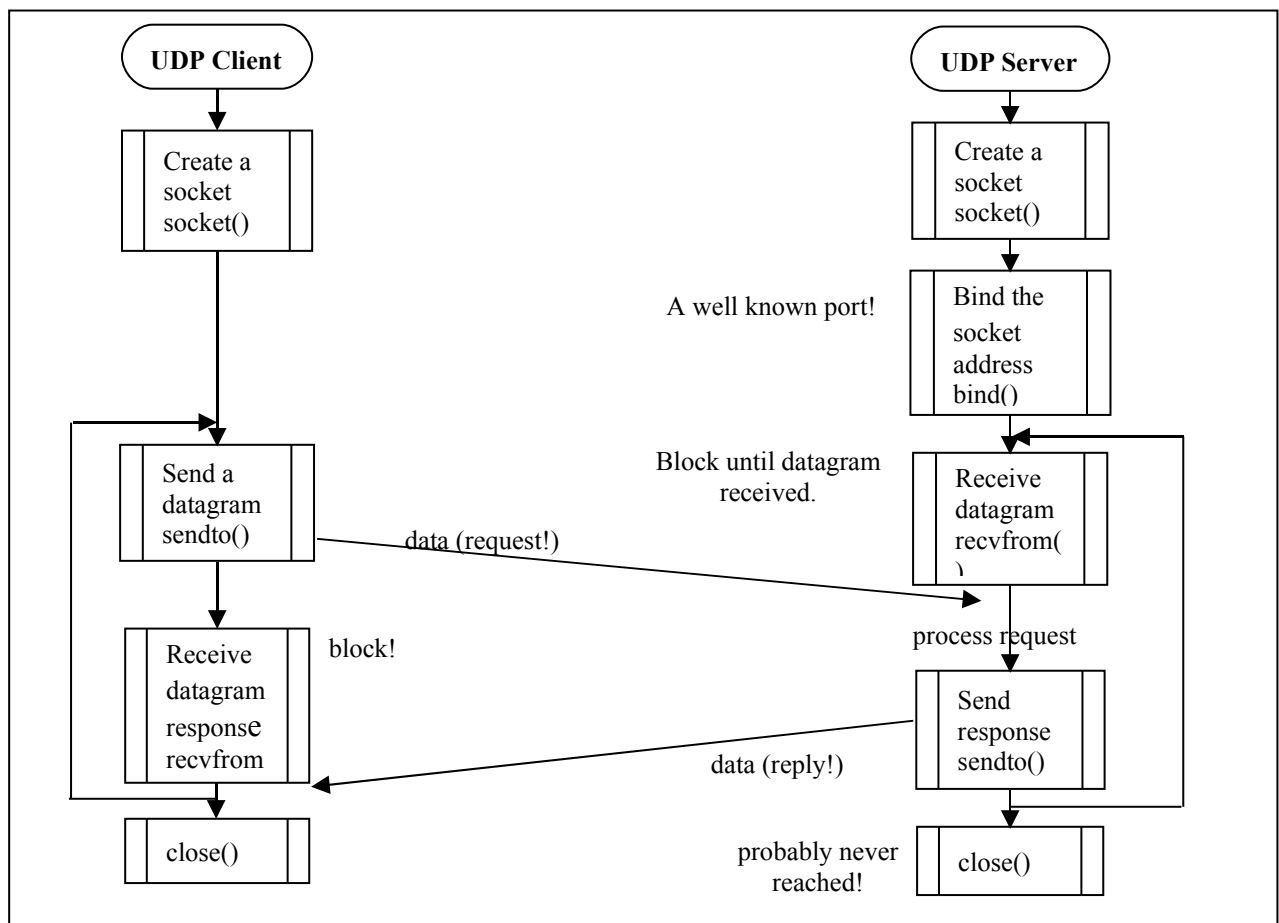
Aims.

The aim of this worksheet is to experiment with sockets using **User Datagram Protocol (UDP)**. It introduces the UDP classes *DatagramSocket* and *DatagramPacket*. It also investigates a simple TCP based ‘web server’.

User Datagram Protocol.

This protocol is often referred to as **Unreliable Datagram Protocol (UDP)** because, unlike TCP streams, there is no guarantee that data sent will reach its destination. When using a TCP socket a connection is established between the end point before communication takes place and this connection is maintained until one end decides to actively close it. With a UDP socket a connection is NOT made, instead the sender just issues a message to its destination and hopes it gets there! The message uses a datagram of fixed length, often termed a record. Since there is no connection between client and server the client can send a datagram to one server and then immediately send a datagram to another server using the same socket! UDP is a *connectionless* protocol.

The client/ server relationship using UDP is shown by the following diagram using the standard C functions.



Notice that, unlike TCP, we do not have the concept of *connect()* and *accept()*. In Java the server doesn't *bind()* either! The client and server are pretty much symmetrical

Let us create a simple UDP client/ server pair where the client just sends a message, which the server displays.

A Simple UDP Server.

In a new window use **an editor** to create the server source code:

```
emacs UDPlistener.java &
```

```
notepad UDPlistener.java
```

Enter the following program:

```
//  
// UDPlistener.java based on code in Dietel and Dietel pp 1004-1007  
// Server reads a datagram packet from a client and responds  
// with a datagram packet.  
//  
import java.io.*;  
import java.net.*;  
  
public class UDPlistener {  
  
    private DatagramSocket mySocket;  
    private DatagramPacket receivePacket;  
    private DatagramPacket sendPacket;  
  
    public UDPlistener() {  
        System.out.println(" \nListening very carefully!\n\n");  
        try {  
            mySocket = new DatagramSocket( 4567);  
        }  
        catch( SocketException sE) { // error so bad server dies!  
            System.err.println( "Could not open a datagram socket");  
            sE.printStackTrace();  
            System.exit( 1);  
        }  
    } // constructor  
    //  
    // Wait for packets from client, display them and respond.  
    //  
    public void waitForPackets() {  
        do { // wait for 'END' from client!!  
            // receive a packet, display it and echo  
            try {  
                // set up a packet ready to receive the msg  
                byte data[] = new byte[ 100];  
                receivePacket = new DatagramPacket( data, data.length);  
  
                // now wait for the packet  
                mySocket.receive( receivePacket);  
  
                displayPacket();  
                sendPacketToClient( "I see you sent: ");  
            }  
            catch( IOException ioE) {  
                System.err.println( "Some receive error!");  
                System.err.println( ioE.toString());  
            }  
        } while( !(receivePacket.toString()).equals( "END"));  
    } // waitForPacket()  
  
    //  
    // Display the received packet  
    //  
    public void displayPacket() {
```

```

    System.out.println( "Packet received: ");
    System.out.println( "\tFrom host: " + receivePacket.getAddress());
    System.out.println( "\tHost port: " + receivePacket.getPort());
    System.out.println( "\tLength: " + receivePacket.getLength());
    System.out.println( "\tContaining: ");
    System.out.println( "\t\t" + new String( receivePacket.getData(), 0,
receivePacket.getLength()));
} // displayPacket()

//
// echo packet
//
private void sendPacketToClient( String msg) throws IOException {

    // construct the packet to send
    String sMsg = msg + " ";
    sMsg += new String( receivePacket.getData(), 0, receivePacket.getLength());
    // create the packet to send
    sendPacket = new DatagramPacket( sMsg.getBytes(), sMsg.length(),
receivePacket.getAddress(), receivePacket.getPort());
    // send the packet
    mySocket.send( sendPacket);
    System.out.println( "Packet sent back");
} // sendPacketToClient()

//
// execute the application
//
public static void main( String args[]) {
    UDPlistener app = new UDPlistener();
    app.waitForPackets();
} // main
} // class UDPlistener

```

Compile the server by:

```
javac UDPlistener.java
```

The server opens a UDP socket on port number 4567, it then initialises a receive *DatagramPacket* of 100 bytes. The server then blocks on *socket.receive()* waiting for a client packet. The received packet is then used to display the sender's IP address and port number. A response, another *DatagramPacket* is sent back. The code begs questions such as 'How do we know the datagram receive packet has been assigned a large enough size'. Once a client is available we can try different packet sizes.

A Simple UDP Client.

For the client we'll try using the *Scanner class* to get keyboard input, rather than the rigmarole of setting a *BufferedReader* object! Using your editor, enter the following client code:

```
emacs UDPclient.java &
```

```
notepad UDPclient.java
```

```

//
// UDPclient.java based on code in Dietel and Dietel pp 1008-1010
// Sends a datagram packet to a server and
// waits for a reply.
//
import java.io.*;
import java.net.*;

```

```

import java.util.Scanner;

public class UDPclient {

    private DatagramSocket mySocket;

    private DatagramPacket sendPacket;
    private DatagramPacket receivePacket;

    public UDPclient() { // constructor

        try {
            mySocket = new DatagramSocket();
        }
        catch ( SocketException sockE) {
            System.err.println( "Problem opening datagram socket!");
            sockE.printStackTrace();
            System.exit( 1);
        }
    } // constructor

    //
    // Send packets, wait for responses from server and display them.
    //
    public void sendPackets() {
        Scanner kbd = new Scanner( System.in);
        String message = null;
        byte rdata[] = new byte[ 200]; // to hold response

        do{ // wait for 'END'
            try {
                // set up datagram packet.
                System.out.println( "\nEnter a message to send (END to disconnect): ");
                // next line of input from keybd.
                message = kbd.nextLine(); // what happens if we use next()?

                if( !message.equals( "END")) { // if more messages to send
                    // set up the packet
                    byte sdata[] = message.getBytes();

                    sendPacket = new DatagramPacket( sdata, sdata.length, InetAddress.getLocalHost(), 4567);
                    mySocket.send( sendPacket);

                    // wait for response!
                    receivePacket = new DatagramPacket( rdata, rdata.length);
                    mySocket.receive( receivePacket);

                    displayPacket();
                } // if msgs to send
            }
            catch( IOException ioE) {
                System.err.println( "Receive error!");
                ioE.printStackTrace();
            }
        } while( !message.equals( "END"));
        System.out.println( "Client finished");
        mySocket.close();
    } // sendPackets()

    //
    // display the received packet
    //
    public void displayPacket() {
        System.out.println( "Packet received: ");
        System.out.println( "\tFrom host: " + receivePacket.getAddress());
        System.out.println( "\tHost port: " + receivePacket.getPort());
        System.out.println( "\tLength: " + receivePacket.getLength());
        System.out.println( "\tContaining: ");
        System.out.println( "\t\t" + new String( receivePacket.getData(), 0,
receivePacket.getLength()));
    } // displayPacket()

```

```

//
// start the application!
//
public static void main( String args[] ) {
    UDPClient app = new UDPClient();

    app.sendPackets();
} // main
} // class UDPClient

```

Compile the client by:

```
javac UDPClient.java
```

Again communication is via *DatagramPackets*. When the client wishes to send, it creates the contents of the packet, including information such as the IP address of the server (at the moment *InetAddress.getLocalHost()*), and the port number on which the server is expecting the datagram. When the client receives a datagram it displays the information contained in the packet. Let's test the relationship between the server (*UDPListener*) and the client (*UDPClient*).

In one window run the server:

```
java UDPListener
```

In another window use **netstat/ps/taskmgr** to check that it is running.

In yet another window run the client:

```
java UDPClient
```

Enter some text and hit enter. Inspect the information displayed by the server (listener) and the client. Do they agree? What port number does the server believe the client transmitted on? Open another window and start a second client. Again enter some text. Is all as expected? Explain the values that the server displays as the port numbers used by the clients. Close the first client (type END). Does the second continue to work? Continue to experiment, when you've finished END each client and kill the server (listener) – use *Ctrl-C!* What happens if the client starts before the server? Is it what you expect?

Modifications.

1. Try changing the datagram buffer size (to 10 say) in the server and re-compile it and re-run the programs. What happens if you send a message longer than this? How might you allow for such messages? Don't forget to change it back to a suitable size!
2. Modify the client so that the server host name and its port number may be supplied on the command line.
3. Modify the client so that the server host name and its port number may be entered as the program runs if they are not given at the command line.

Once the client has been modified to let the user supply the server name and port number try your client against someone else's server (listener).

A Simple Web Server using TCP.

Just to show that we can do something 'useful' with our sockets try this simple Web Server! It uses TCP sockets and can interact with Web Browsers such as Netscape, Mozilla and Internet Explorer.

emacs Web8000.java

notepad Web8000.java

```
//
// Web8000.java an HTTP Web server from Elliott Rusty Harold.
// Listens on a socket waiting for a client to send a GET request.
// Always responds with the contents of a fixed text file!
//
import java.io.*;
import java.net.*;
import java.util.*;

public class Web8000 extends Thread {
    private byte[] header;    // the message header
    private byte[] content;  // the message content

    private final static int DEFAULT_PORT = 8000; // the default listening port
    private int port = DEFAULT_PORT;

    // constructor
    public Web8000( byte[] data, String encoding, String MIMETYPE, int port)
        throws UnsupportedOperationException {
        content = data;                // set up the message content
        this.port = port;              // use the port number passed
        String header = "HTTP/1.0 200 OK\r\n"
            + "Server: Web8000 1.0\r\n"
            + "Content-length: " + content.length + "\r\n"
            + "Content-type: " + MIMETYPE + "\r\n\r\n";
        // note the extra '\r\n', see what happens if it is missing!
        this.header = header.getBytes( "ASCII"); // assume ASCII encoding system
    } // constructor

    // enable thread to deal with client
    public void run() {
        try {
            ServerSocket serverSock = new ServerSocket( port);
            // log details of server response on screen
            System.out.println( "Connecting on port: " + serverSock.getLocalPort());
            System.out.println( "Data to be sent: ");
            System.out.write( content);

            while( true) { // loop forever dealing with
clients
                Socket clientSock = null;
                try {
clients
                    clientSock = serverSock.accept(); // get skt for client-server
clients
clients
                OutputStream out = new BufferedOutputStream( clientSock.getOutputStream());
                InputStream in = new BufferedInputStream( clientSock.getInputStream());

                // read request and assume buffer big enough!
                StringBuffer request = new StringBuffer( 800);
                int ch = 0;
                Boolean finished = false; // so we can read bytes until all of hdr.

                while( !finished) {
                    ch = in.read(); // read next char (byte), actually returns int!
                    if( ch == -1) { // did sender somehow fail during transmission?
                        System.out.println( "Sender died?");
                    }
                }
            }
        }
    }
}
```

```

        finished = true;
    }
    else {
        // append the char to the string buffer
        System.out.print( (char)ch);
        request.append( (char)ch);
        if( ch == '\n') { // see if end of header
            if( request.toString().indexOf( "\r\n\r\n") != -1)
                finished = true; // what if "\r\r" or "\n\n"?
        }
    } // deal with bytes recvd
} // while more bytes to read

// check that it is an HTTP request. Should check for GET etc too!
if( request.toString().indexOf("HTTP/") != -1)
    out.write( header); // okay so write MIMEtype hdr details
// now send the actual data content
out.write( content);
out.flush(); // to make sure the buffer is emptied immediately
} // try
catch( IOException ioE) {
}
finally { // make sure this end of the connection gets closed
    if( clientSock != null) clientSock.close();
}
} // while forever
} // try opening server socket
catch( IOException ioE) {
    System.err.println( "Could not start server. Port in use!");
}
} // run()

public static void main( String args[]) {
    try {
        String fileName = "data.txt"; // should be supplied by client!
        String contentType = "text/plain"; // make this depend on file extension
        // e.g. .html -> "text/html"

        InputStream in = new FileInputStream( fileName); // open file
        // array to hold contents of file
        ByteArrayOutputStream out = new ByteArrayOutputStream();

        int bite;
        while( (bite = in.read()) != -1) // Now read bytes from file ...
            out.write( bite); // ... into byte stream
        // convert byte stream into an array of bytes ready for DatagramPacket
        byte [] data = out.toByteArray();

        int port = DEFAULT_PORT;
        String encoding = "ASCII";

        // create the thread to send the contents of the file to any connection!
        Thread thr = new Web8000( data, encoding, contentType, port);
        thr.start(); // start the thread to deal with the clients
    } // try
    catch( ArrayIndexOutOfBoundsException obE) {
        // useful for when args should have been supplied!
        System.err.println( "Usage: java Web8000 filename port encoding");
    }
    catch( Exception e) {
        System.err.println( e);
    }
} // main

```

```
} // class Web8000
```

This Web server doesn't do much. In response to a client request it returns the contents of a known file, *data.txt*. You must create the file *data.txt* using an editor, e.g. the file contents might be:

```
Hello from Web8000
How are you?
Hope you get this message!
```

Compile the web server with:

```
javac Web8000.java
```

Run the server and test it by using **telnet**:

```
telnet localhost 8000
```

We are using port number 8000 not the standard port 80!

We need to enter a simple HTTP request so try:

```
GET /name.html HTTP/1.1
```

Don't forget to also enter a blank line after this one to show end of HTTP header!

The server does not check the file name requested it just returns the standard we created! What happens if either GET or HTTP is in lower case, or mixed case or we use just / instead of /name.html?

Once satisfied that the server works start a web browser (Mozilla or IE say) and enter a URL, e.g.

```
http://localhost:8000/
```

Note that we must supply the server hostname and the port number the server uses.

Does it work?

We should see the contents of the data file displayed in the browser and our server should display the contents of the header message it received! (The end of the header is indicated by a blank line, i.e. "\r\n\r\n". Edit *Web8000.java* to send just '\r\n' and see what happens!) Does Internet Explorer send the same request message as Mozilla?

Try connecting to the simple web server of a colleague.

Try browsing for `http://localhost:8000/index.htm`. The browser might give a different display it all depends on how rigorously it interprets the HTTP protocol! What happens if you check for 'http/' instead of 'HTTP/'?

Change the text file to be `html` by adding:

```
<html><body><h1>
```

at the beginning , and:

```
</h1></body></html>
```

at the end. Does the browser manage to display as expected?

Notice that this program assumes an encoding method of 'ASCII'. What other encoding methods are available?

Try modifying **Web8000** so that the port number and encoding method are command line args. Modify the program so that it can supply the contents of different files. Allow for 'File not Found' by setting up a proper HTTP response so that the browser can display the standard 404 message!

Try to write a simple client program that accepts a file name and server IP address and port number. It should then create a header like the one generated by your browser and send it to the server. Any response cannot be rendered properly so just display the characters received.

Finally.

This has been just a taste of the functions available for socket programming. Have fun experimenting with these programs.

There is a lot of information on the internet and many books published on the subject. Look around try things out, don't be afraid to experiment more than we have done here.

References.

Java – How to Program, 4th Edition, Deitel and Deitel, Prentice-Hall.

Java Network Programming, 2nd Edition – Elliotte Rusty Harold, O'Reilly.

Sun Java tutorials.

Unix Network Programming, Vol. 1. 2nd Edn., W. Richard Stevens, Prentice-Hall Pearson Education.