Java Multi-threaded Webserver

Aren Tyr

(Updated September 2019)

1 Introduction

This program explores the concepts of multithreading and the client-server model in Java.

The Java API provides an extremely rich and featureful development environment; consequently, extremely powerful applications that could take considerable amounts of code in other languages (such as C) become straightforward and concise in Java.

Java provides particularly good mechanisms for dealing both with concurrent threads of execution and with Network interfaces. Indeed, the plethora of options available to the Java developer is almost overwhelming in its comprehensiveness – fortunately, the HTML version of the API is both convenient and quick to access.

Initially I set no *particular* aim for what my client-server pair would do. I decided that I would concentrate the majority of the development on the server and see what system I could develop as I experimented with the API.

However, the ease of development under Java naturally allowed a progression of complexity, until I ended up implementing an entire multithreaded web server, as presented here. Since there are many existing web browsers, I decided it would be more interesting to attempt to write a server that could serve pages to these clients, rather than work on some more limited Java-only client-server pair.

2 Design

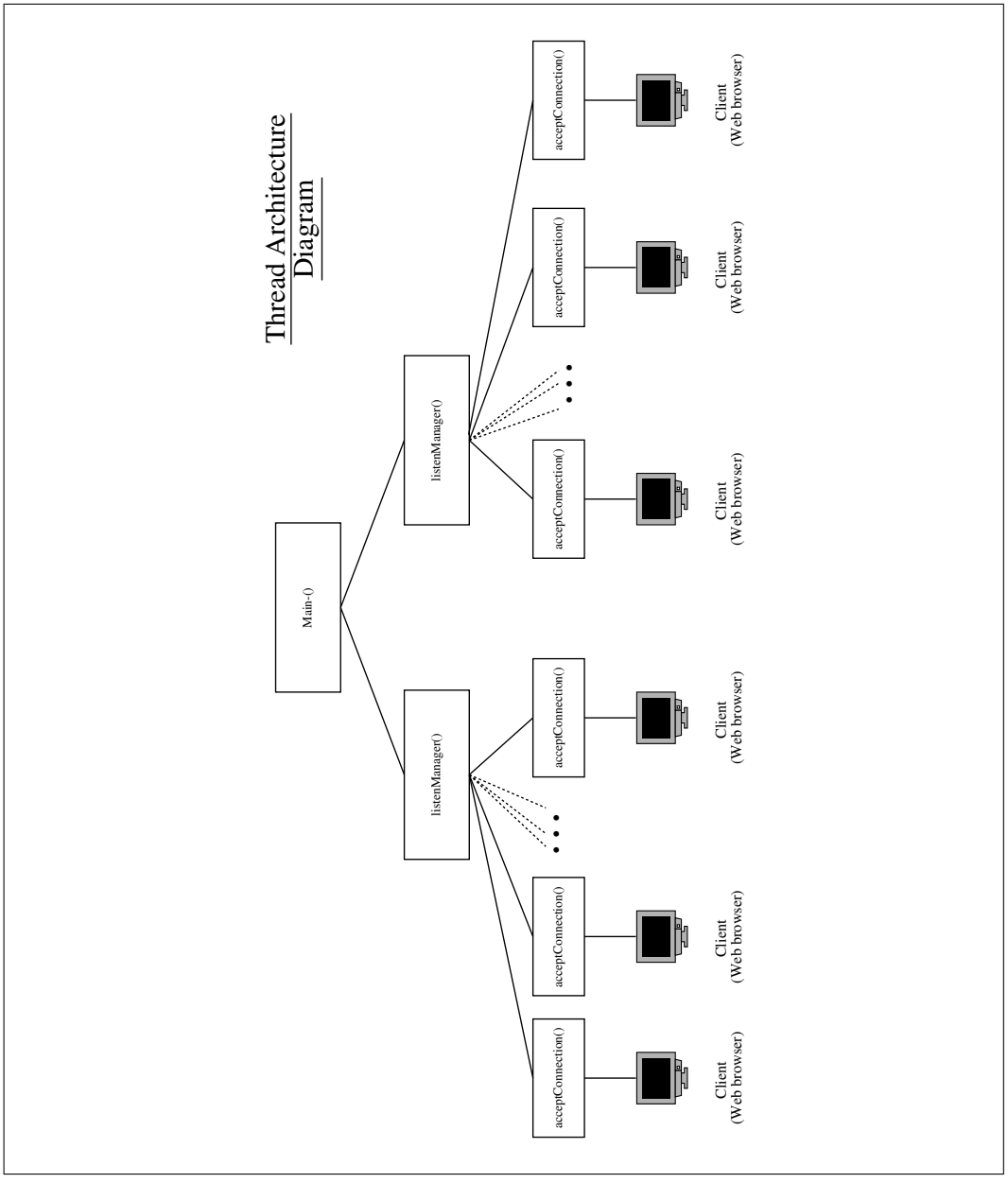
The client and server communicate by exchanging UTF-8 encoded strings via `writeUTF()`. This relatively high-level method to network transmissions enables communication over the network without having to explicitly transmit header or other meta-information – Java automatically handles it for you. At this early stage I had already formulated the thread architecture I was planning on using for my Server. I opted to use a listener thread that would spawn new threads per `accept()` request as necessary to service the request. However, I also decided that I would use *two* listener threads to potentially increase the throughput of the system.

By default, if multiple `accept()` requests occur simultaneously, they will be put into a queue and serviced sequentially (the default queue length, I believe, is around 50).

I hypothesised that creating two listener threads would reduce the latency of thread creation[1], and help to prevent avoid overflowing the internal queue – since there are now two queues – when the server is under extremely heavy loads and the request events are being generated faster than the listener thread is able to spawn new threads to service them.

The figure below depicts this thread architecture that was consequently used by my web server:

[1] Though admittedly, possible thread contention issues arise that might compromise maximum performance.



After creating an initial client-server pair, I decided to exercise the multithreading capabilities of the server by constructing a modified client program that would spawn many threads that would simultaneously attempt to connect to my server. By putting a deliberate, random `Thread.sleep()` delay in the thread on the Server servicing the request, I could see if it was in fact actually multithreading. Using this, I was able to adjudge that the multithreading was working. At this stage, I had not entirely pre-planned what my server would actually

do. So I decided that it would be nice if instead of simply exchanging UTF-8 encoded strings between the Java client and server, my Server could serve up a simple HTML page to any given web browser.

When a web browser attempts to connect to any given server, it issues at `GET` request for the named resource. The Server will read in the request, and then deliver the response. If you telnet to port 80 on any web server, and issue, for example `GET /index.html`, then assuming the page exists, the web browser will return the page *preceded* by a HTTP protocol header.

Whilst I had some familiarity with this header, I was not sure of the exact details, so I looked up the explicit technical details from the W3C[2] consortium's web page. The RFC documents on their site, although terse, are both comprehensive and exact.

One important detail of the header specification is that each line must be terminated by a carriage return line feed, `CrLF (\n\r)`. Any given web client expects this when differentiating the lines of the HTTP header that defines the HTTP protocol. Perhaps the most important insight I had was to use the header line "`Content-encoding:chunked`", since this makes the "`Content-length:`" field unnecessary – otherwise, you have to specify the exact length (in bytes) of the document you're sending to the client – which is tiresome and awkward.

Once I had the HTTP protocol strings determined, it was simply a question of delivering the strings via a suitable output stream. `PrintStream` worked fine for me, though no doubt other output streams could also be used – since Java has a multitude of wrappers for both data input and output.

In terms of actually reading the initial request from the browser, there are various methods available as part of the `DataInputStream` class. Simply reading the request into a fixed sized buffer sufficed.

Since you need to tell the browser the MIME type of the material you're sending, via the "`Content-type:`" header, it was necessary to parse the browser's request, and determine the MIME type required based on the filename extension. HTML normally ends with either .html or .htm, so the header would be "`Content-type: text/html`" and correspodingly, for GIF image files, ending with .gif, the header would be "`Content-type: image/gif`". For every standard type of file, their is an according MIME type. I only explicitly programmed a few major types into my server (HTML, JPG, PNG, GIF, and Unknown).

Combining all this, I was able to deliver an explicit HTML page as strings. Finally, I simply modified it so that it could read the given page to deliver (if it exists) from the specified server root directory (I set it to `./html`) and transmit this to the browser.

Rather than use a fixed buffer for reading the file into – since you cannot predetermine the necessary size of buffer to use – I wrote it so that it actually uses a four kilobyte *pipe* to transmit the data. When (up to) 4k has been read in from the file, this data is transmitted.

[2]The W3C is the closest organization that the Internet has for standards defining body. For full standards-compliance on the internet, W3C is the primary source.

It then reads in the next 4k into the buffer and transmits this. This cycle continues until we have hit the end of a file (the integer file descriptor used whilst reading data from the `InputStream` will indicate this by being set to the value -1) where we output the final CrLF and hopefully the web browser will display our transmitted page correctly.

When a web browser is downloading a HTML page with multiple items – several frames, images, etc. – it will open a new socket to the web server to obtain the given file (e.g. an image file) and send the according GET request. By determining the MIME type as discussed above, we have in effect automatically enabled the server able to deal with pages of arbitrary complexity, since the page is transmitted in response to nothing more than a series of GET requests.

I now had a fully multithreaded web server. The server is admittedly relatively primitive in many respects – there is not much security in the web server, there is no support for any of the more advanced web technologies (such as PHP for example), and various configuration settings are explicitly coded (rather than being parsed from a configuration file).

Nevertheless, this was only intended as a "proof-of-concept" learning exercise, so I am extremely happy with the final result [3].

[3] And my server is infinitely more simple to configure than Apache :-)

3 Coding

Here is the code of the simple client:

```
/* Filename:  Client.java
 * Author:    Aren Tyr
 * Date:      24/09/19
 *
 * An extremely simple client that simply reads a
 * String from the Server. */

import java.net.*;
import java.io.*;
import java.util.*;                                                    10



public class Client
{
        public static void main(String[] args)
        {
                Socket sockConnect;
                int portOpen = 0;
                InputStream inputData;                                 20
                DataInputStream recieveData;

                try
                {
                        portOpen = Integer.parseInt(args[1]);
                }
                catch(NumberFormatException e)
                {
                        System.out.println(e.getMessage());
                        System.exit(1);                                30
                }

                try
                {

                        sockConnect = new Socket(args[0], portOpen);
                        System.out.println("Got connection.");

                        inputData = sockConnect.getInputStream();
                        recieveData = new DataInputStream(inputData);  40

                        //read data from Server
                        String message = recieveData.readUTF();
                        System.out.println("Client recieved \"" + message + "\" from server.");

                        recieveData.close();
                        inputData.close();

                        sockConnect.close();
                                                                       50
                }
                catch(IOException e)
                {
```

```
                System.out.println(e.getMessage());
            }
        }
}
```

Here is the modified version of the above file that creates 100 clients all attempting to simultaneously connect:

```
/* Filename:  ClientThreadTest.java
 * Author:    Aren Tyr
 * Date:      24/09/19
 *
 * This is test program to exercise the multithreading capabilities
 * of the server by spawning off 100 threads all trying to obtain
 * data from the server. */

import java.net.*;
import java.io.*;                                                    10
import java.util.*;

public class ClientThreadTest
{
        public static void main(String[] args)
        {
                //create an array of Threads
                Thread p[] = new makeRequest[100];

                                                                     20
                try
                {
                        //fire off all the Threads
                        for(int i=0; i<100; i++)
                        {
                                p[i] = new makeRequest();

                                p[i].start();
                        }
                }                                                    30
                catch(Exception e)
                {
                        System.out.println(e.getMessage());
                        e.printStackTrace();
                }
        }
}

class makeRequest extends Thread
{                                                                    40

        public void run()
        {
                Socket sockConnect;

                InputStream inputData;
                DataInputStream recieveData;
```

```
            String p = "localhost";

            try
            {
                sockConnect = new Socket(p, 8100);
                System.out.println("Got connection.");

                inputData = sockConnect.getInputStream();
                recieveData = new DataInputStream(inputData);

                //read the data from the server
                String message = recieveData.readUTF();
                System.out.println("Client recieved \"" + message + "\" from server.");

                recieveData.close();
                inputData.close();

                sockConnect.close();

            }
            catch(IOException e)
            {
                System.out.println(e.getMessage());
            }
        }
}
```

Here is the code of the initial simple server I constructed:

```
/* Filename:  simpleServer.java
 * Author:    Aren Tyr
 * Date:      23/09/19
 *
 * A simple multithreaded server that listens for requests
 * and then spawns a new thread to service the request.
 */


import java.net.*;
import java.io.*;
import java.util.*;

//Main program class
public class simpleServer
{

        //number of active connections to our server
        static int numberOfConnections = 0;

        //increment connections
        static void incConnections()
        {
                ++numberOfConnections;
```

```
}

//decrement connections
static void decConnections()
{
        --numberOfConnections;                                                  30
}

//return the number of active connections
static int retConnections()
{
        return numberOfConnections;
}

public static void main(String[] args)
{                                                                               40

        //create a ServerSocket object
        ServerSocket servSocket = null;
        int portConnect = 0;

        System.out.println("Attempting to start server...");

        // Attempt to determine to port to start the server on
        try
        {                                                                       50
                portConnect = Integer.parseInt(args[0]);
        }
        catch(NumberFormatException e)
        {
                System.out.println("Invalid value specified for port.");
                System.exit(1);
        }

        // Attempt to create a system socket on the given port
        try                                                                     60
        {
                servSocket = new ServerSocket(portConnect);
        }
        catch(IOException e)
        {
                System.out.println("Unable to initialize port. Is the port restricted?");
                System.exit(1);
        }


                                                                                70
        //main program execution

        listenManager p1 = new listenManager(servSocket);
        listenManager p2 = new listenManager(servSocket);

        /* fire off two threads, each listening -
         * the aim is to allow the server to maximize
         * the number of possible simulateous connections. */
        p1.start();
        p2.start();                                                             80
```

```
                System.out.println("Server started. Listening on port: " + Integer.parseInt(args[0]));

        }
}

/* This class listens for requests and spawns off
 * threads accordingly to deal with the request.
 */
class listenManager extends Thread                                              90
{
        ServerSocket localSock = null;

        Socket mSocket = null;

        public listenManager(ServerSocket sS)
        {
                localSock = sS;
        }
                                                                                100
        public void run()
        {
                for(;;)
                {
                        try
                        {
                                //accept the connection attempt on the socket
                                mSocket = localSock.accept();

                                //increment the number of active connections    110
                                simpleServer.incConnections();

                                //give the current connection a unique name
                                System.out.println("Got connection: " + simpleServer.retConnections());

                                //create a connectionManager thread
                                acceptConnection connectionManager = new acceptConnection(mSocket);
                                //now deal with the request
                                connectionManager.start();
                        }                                                       120
                        catch(IOException e)
                        {
                                System.out.println(e.getMessage());
                        }
                }
        }
}

/* This class actually services the request by sending
 * the HTML code back to the client                                             130
 */
class acceptConnection extends Thread
{
        //create the necessary network objects
        Socket opSocket;
        OutputStream mainOutput;
        DataOutputStream sendOutput;
```

```
        public acceptConnection(Socket socketConnection)
        {
                opSocket = socketConnection;
        }

        void negotiateTransfer()
        {

                try
                {

                        //delay for an random amount of time to test
                        //that the mulithreading is actually working
                        Thread.sleep((int)Math.random() * 1000);
                        mainOutput = opSocket.getOutputStream();
                        sendOutput = new DataOutputStream(mainOutput);

                        //output OK & the current timestamp
                        sendOutput.writeUTF("OK" + (new Date()).toString());

                        sendOutput.close();
                        mainOutput.close();
                        opSocket.close();

                        //decrement the number of active connections

                        simpleServer.decConnections();
                }
                catch (IOException e)
                {
                        System.out.println("Transfer attempt failed.");
                }
                catch (Exception e) {};

        }


        public void run()
        {

                //attempt to service the connection request
                negotiateTransfer();

        }
}
```

And here is the finished multithreaded web server:

```
/* Filename:  Server.java
 * Author:    Aren Tyr
 * Date:      24/09/19
 *
 * Description:
 *
 * This program implements a fully mulithreaded webserver capable of serving
```

```
 * up HTML pages and any other arbitrary MIME types over the HTTP protocol.
 *
 * The MIME-type string definitions simply need to be added for them to be
 * served - e.g. "video/mpeg".
 *
 */


//import the necessary Java packages
import java.net.*;
import java.io.*;
import java.util.*;

//Main program class
//*************************************************************************
public class Server
{
	//number of current connections to server
	private static int noOfConnections = 0;

	//increment connections
	static void incConnections()
	{
		noOfConnections++;
	}

	//decrement connections
	static void decConnections()
	{
		noOfConnections--;
	}

	//return the number of active connections
	static int retConnections()
	{
		return noOfConnections;
	}

	public static void main(String[] args)
	{

		//create a ServerSocket object
		ServerSocket servSocket = null;
		int portConnect = 0;

		System.out.println("Attempting to start server...");

		// Attempt to determine to port to start the server on
		try
		{
			portConnect = Integer.parseInt(args[0]);
		}
		catch(NumberFormatException e)
		{
			System.out.println("Invalid value specified for port.");
			System.exit(1);
		}
```

```
                // Attempt to create a system socket on the given port
                try
                {
                        servSocket = new ServerSocket(portConnect);
                }
                catch(IOException e)
                {
                        System.out.println("Unable to initialize port. Is the port restricted?");
                        System.exit(1);
                }


                //main program execution

                listenManager p1 = new listenManager(servSocket);
                listenManager p2 = new listenManager(servSocket);

                /* fire off two threads, each listening -
                 * the aim is to allow the server to maximize
                 * the number of possible simulateous connections. */
                p1.start();
                p2.start();

                System.out.println("Server started. Listening on port: " + Integer.parseInt(args[0]));
        }
}

/* This class listens for requests and spawns off
 * threads accordingly to deal with the request.
 */
//*****************************************************************************
class listenManager extends Thread
{
        private ServerSocket localSock = null;

        private Socket mSocket = null;

        public listenManager(ServerSocket sS)
        {
                localSock = sS;
        }

        public void run()
        {
                for(;;)
                {
                        try
                        {
                                //accept the connection attempt on the socket
                                mSocket = localSock.accept();

                                //increment the number of active connections
                                Server.incConnections();

                                int connections = Server.retConnections();
```

```
                        //give the current connection a unique name
                        String conS = String.valueOf(connections);
                        String r = "Connection" + conS;

                        //create a connectionManager thread
                        acceptConnection connectionManager = new acceptConnection(mSocket, r);
                        //now deal with the request
                        connectionManager.start();                                          130
                    }
                    catch(IOException e)
                    {
                        System.out.println(e.getMessage());
                    }
                }
        }
}

/* This class actually services the request by sending                                     140
 * the requested file back to the client
 */
//*************************************************************************
class acceptConnection extends Thread
{

        // *** Create the necessary network objects ***

        // ------    Output ----------
        private Socket opSocket;                                                            150
        private OutputStream mainOutput;
        private DataOutputStream sendOutput;
        private PrintStream printHTML;

        //------    Input ----------
        private InputStream iS;
        private BufferedInputStream bIS;
        private DataInputStream diS;
        private File dataToServe;
                                                                                            160
        // ****************************************

        // flow control booleans

        private boolean matchedPage = false;
        private boolean dataLeft = true;
        private boolean headerControl = false;
        // Strings used for processing requests

        private String connectionName;                                                      170
        private String pageRequest = "";
        private String pageRoot = "./html/";

        // *** HTTP Protocol Header strings **********
        // Request OK:
        private String headerOK = "HTTP/1.1 200";
        // The MIME type to send to the browser
        private String headerTYPE = "Content-type:";
```

```
// Encoding format (chunked allows dynamic length)
private String headerENC = "Content-Encoding:chunked";                    180

// MIME type (e.g. text/html) string
private String mtype = "";

//This creates the CRLF (carriage return line feed) termination
//symbol needed by the HTTP protocol
private static final byte[] LINETERM = {(byte)'\r', (byte)'\n' };

// ****************************************
                                                                         190
// Byte arrays for storing requests & sending data

private byte buffer[] = new byte[4096]; //4k buffer for reading request into
private byte webpage[] = new byte[4096]; //4k pipe for transmitting data

private static final int MAXFILELENGTH = 256;

//lookup table for MIME types for use in the "Content-type" field
private static final String[] MIMEtypes = {"text/html", "image/jpeg", "image/gif",
                            "image/png", "text/plain", "unknown/unknown"};   200


// other instance variables
private int useIndex = 0;
private int readBytes = 0;
private int j = 1;
private int readData = 0;

public acceptConnection(Socket socketConnection, String conName )
{                                                                        210
        opSocket = socketConnection;
        connectionName = conName;
}

void obtainStreams()
{
        try
        {
                //attempt to get the input stream for the socket
                iS = opSocket.getInputStream();                          220
                mainOutput = opSocket.getOutputStream();

                //output the connection name on STD I/O
                System.out.println("Got connection : " + connectionName);

                //create a new printStream object for writing
                printHTML = new PrintStream(mainOutput);

                //read the browser's request - we don't actually
                //do anything with it                                    230
                readBytes = iS.read(buffer);

        }
        catch (IOException e)
        {
```

```
                System.out.println("Transfer attempt failed.");
        }
        catch (Exception e) {};
}
                                                                                240
void acceptConnection()
{

                for(int i=0; i < readBytes; i++)
                {
                        char p = (char)buffer[i];
                        //check to see if we have a GET request...
                        if((p == 'G' || p == 'E' || p == 'T' || p == ' ') && headerControl == false)
                                continue;
                                                                                250
                        //determine the requested file
                        else if(p == '/' && headerControl == false)
                        {
                                headerControl = true;

                                //the case where nothing (i.e. the server root) was requested
                                if((char)buffer[i+j] == ' ')
                                {
                                        pageRequest = "index.html";
                                        mtype = "html";                         260
                                        matchedPage = true;
                                        break;
                                }


                                //parse the input until we either have a space (bad)
                                //or a period, signifying that we should look at the
                                //file extension
                                while(((char)buffer[i+j] != ' ') && ((char)buffer[i+j] != '.'))
                                {                                               270
                                        pageRequest += (char)buffer[i+j];
                                        j++;

                                        if(j > MAXFILELENGTH)
                                        {
                                                System.out.println("Filename request is too large.");
                                        }

                                }
                                                                                280
                                //parse the file extension to determine the MIME type
                                if(((char)buffer[i+j] == '.') && (readBytes > (buffer[i+j]+5)))
                                {
                                        while((char)buffer[i+j] != ' ')
                                        {
                                                if((char)buffer[i+j] != '.')
                                                        mtype = mtype + (char)buffer[i+j];

                                                pageRequest = pageRequest + (char)buffer[i+j];
                                                j++;                            290
                                                matchedPage = true;
                                        }
```

```
                        }
                    }

                }

}
                                                                                    300
void determineMIMEtype()
{
                //determine which "Content-type: <MIME type>" string to use.
                //useIndex is an index into our MIMEtypes lookup table
                if(mtype.equalsIgnoreCase("html") || mtype.equalsIgnoreCase("htm"))
                        useIndex = 0;
                else if(mtype.equalsIgnoreCase("jpeg") || mtype.equalsIgnoreCase("jpg"))
                        useIndex = 1;
                else if(mtype.equalsIgnoreCase("gif"))
                        useIndex = 2;                                               310
                else if(mtype.equalsIgnoreCase("png"))
                        useIndex = 3;
                else if(mtype.equalsIgnoreCase("txt"))
                        useIndex = 4;
                else
                        useIndex = 5;
}


void printHTTPheader()                                                              320
{
        try
        {
                //send the HTTP header to the browser
                // *** HTTP HEADER ****
                printHTML.print(headerOK);
                printHTML.write(LINETERM);
                printHTML.print(headerTYPE + MIMEtypes[useIndex]);
                printHTML.write(LINETERM);
                printHTML.print(headerENC);                                         330
                printHTML.write(LINETERM);
                printHTML.write(LINETERM);
                // ********************
        }
        catch(IOException e)
        {
                System.out.println("An error whilst sending the HTTP header.");
        }
        catch(Exception e)
        {                                                                           340
                e.printStackTrace();
        }
}


void sendData()
{

        try
```

```
        {                                                                                    350

                //try to read the requested file
                diS = new DataInputStream(new BufferedInputStream(new FileInputStream(dataToServe)));
                do
                {
                        /*
                         * Create a data pipeline.
                         *
                         * Read up to 4k of the file in at a time before sending
                         * it until the file is completely read.                           360
                         */

                        readData = diS.read(webpage, 0, webpage.length);

                        if(readData != -1)
                        {
                                for(int k = 0; k < readData; k++)
                                printHTML.write(webpage[k]);
                        }
                                                                                            370
                        //file has been completely read...
                        if(readData == -1)
                        {
                                dataLeft = false;
                                printHTML.write(LINETERM);
                        }


                } while (dataLeft == true);
                                                                                            380

        }
        catch(IOException e)
        {
                System.out.println("Error reading webpage file.");
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }                                                                                   390
}


void serveRequest()
{
        //attempt to obtain a file descriptor for the given request
        dataToServe = new File(pageRoot + pageRequest);

        //does the requested page exist? If yes...
        if(dataToServe.exists() == true)                                                    400
        {
                System.out.println(" -> MIME type requested: " + MIMEtypes[useIndex]);
                System.out.println(" -> Item " + pageRequest + " requested - serving...");
                printHTTPheader();
                sendData();
```

```
            }
            //page doesn't exist, so output error.html to browser
            else
            {
                System.out.println("Request for non-existant page.");
                dataToServe = new File(pageRoot + "error.html");
                useIndex = 0;
                printHTTPheader();
                sendData();
            }
        }

        void closeStreams()
        {
            //tidy up
            try
            {
                printHTML.close();
                iS.close();
                mainOutput.close();
                opSocket.close();
            }
            catch(IOException e)
            {
                System.out.println("Closing streams cleanly failed.");
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }

        void negotiateTransfer()
        {

            obtainStreams();
            acceptConnection();
            determineMIMEtype();
            serveRequest();
            closeStreams();

            //decrement the number of active connections
            Server.decConnections();
        }


        public void run()
        {

            // MAIN THREAD EXECUTION STARTUP
            //attempt to service the connection request
            negotiateTransfer();

        }
}
```

4 Testing

4.1 simpleServer.java

Here is the output – condensed for presentation here – from the simple server after starting it on port 8100 and then starting off the ClientThreadTest program that creates 100 threads that attempt to connect to the server (all running on my local machine):

```
=~/documents/src/client-server=>java simpleServer 8100
Attempting to start server...
Server started. Listening on port: 8100
Got connection: 1
Got connection: 2
Got connection: 3
Got connection: 2
Got connection: 1
Got connection: 1
Got connection: 2
Got connection: 2
Got connection: 1
Got connection: 1

...

Got connection: 4
Got connection: 1
Got connection: 1
Got connection: 1
```

4.2 ClientThreadTest.java

And the corresponding output from the `ClientThreadTest` program:

```
=~/documents/src/client-server=>java ClientThreadTest localhost 8100
Got connection.
Got connection.
Client recieved "OKSun Sep 29 11:03:48 GMT 2019" from server.
Got connection.
Client recieved "OKSun Sep 29 11:03:48 GMT 2019" from server.
Client recieved "OKSun Sep 29 11:03:48 GMT 2001" from server.

...

Got connection.
Got connection.
Got connection.
Got connection.
Client recieved "OKSun Sep 29 11:04:54 GMT 2019" from server.
```

```
Client recieved "OKSun Sep 29 11:04:54 GMT 2019" from server.
Client recieved "OKSun Sep 29 11:04:54 GMT 2019" from server.
Client recieved "OKSun Sep 29 11:04:54 GMT 2019" from server.
```

4.3 Server.java

Figure 1 shows that my web server is correctly serving up this basic HTML page – notice the URL in the browser.

This is the corresponding output from my web server – you can clearly see the order in which the various elements of the page have been requested by the client browser:

```
=~/documents/src/client-server=>java Server 8100
Attempting to start server...
Server started. Listening on port: 8100
Got connection : Connection1
-> MIME type requested: text/html
-> Item index.html requested - serving...
Got connection : Connection1
-> MIME type requested: image/gif
-> Item index_html_6ad845b73d23748b.gif requested - serving...
```

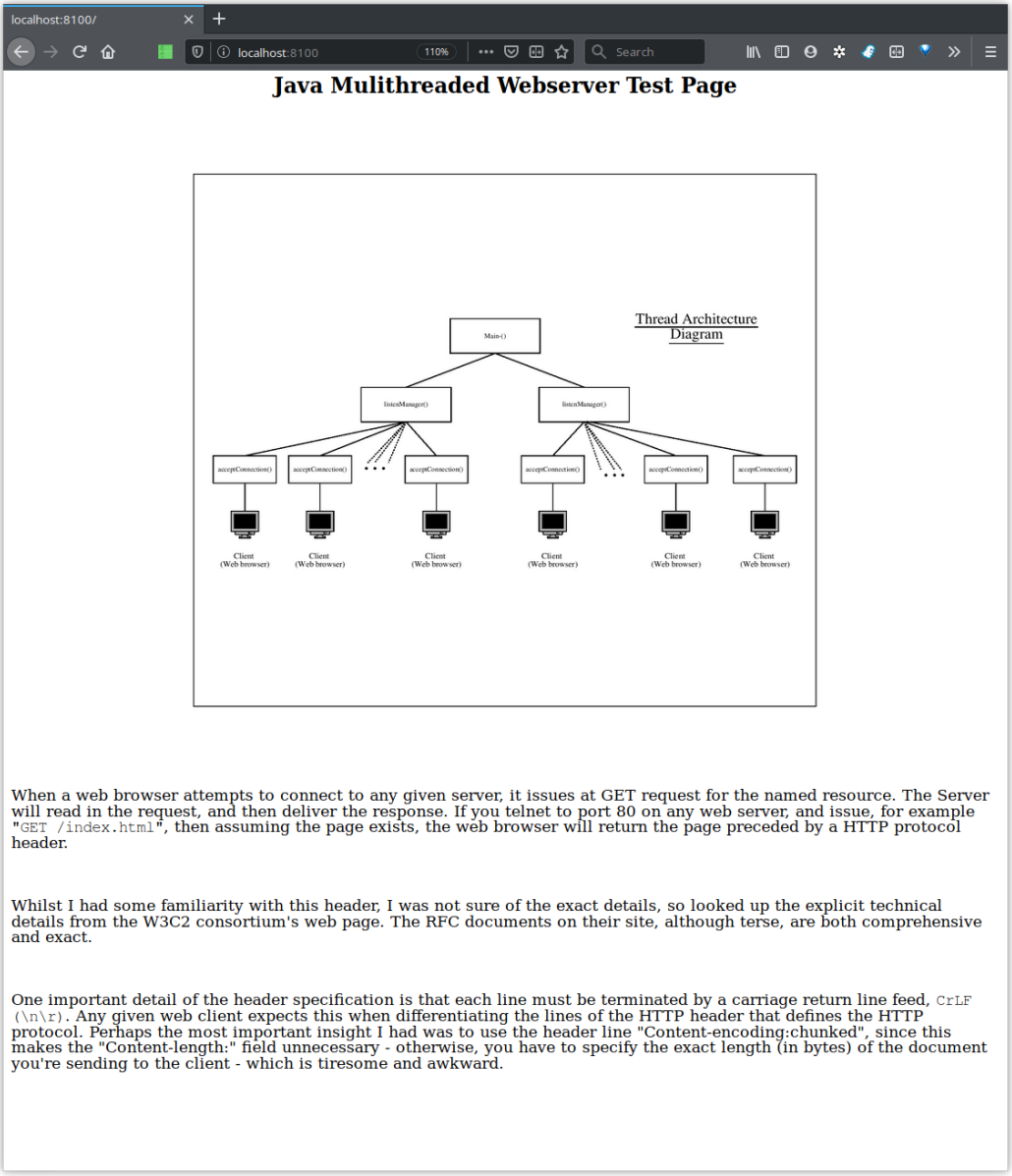


Figure 1: A basic test page.

If you request a non-existent page it will output a warning page, as figure 2 shows.
Finally, as figure 3 shows, it can handle arbitrary MIME types – if the type is unknown, it will output it with type `unknown/unknown` allowing the user to download the file from their web browser.

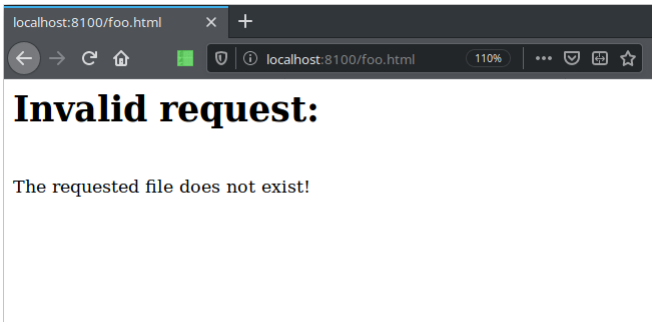


Figure 2: Requesting a non-existent resource.

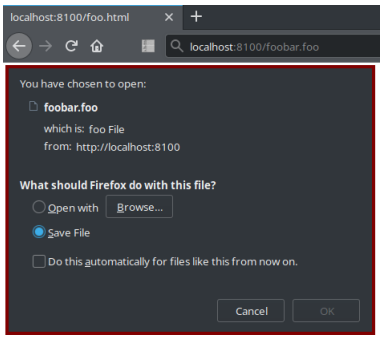


Figure 3: Unknown/alien MIME type request.

5 Conclusion

Overall I'm satisified with my web server. It works reasonably efficiently, and development with the Java Networking API was rewarding.

6 Acknowledgements

This document was fomatted with LaTeX, using the additional module *lgrind* to "pretty-print" the souce code. The diagram was drawn using Dia.

7 References

My main soure of reference for this project was, of course, the Java API documentation.

The following site was of particular help:

- `http://www.w3.org/Protocols/` for the RFC document on the HTTP protocol.