Overview

This assignment will introduce you to programming network applications with Java using threads. You will also learn about HyperText Transfer Protocol (HTTP), which is the protocol that you use when you browse the web. You will do this by **writing a simple HTTP server**. The web server you write will not be as featured as the web servers used on the Internet, though it may become clear how you would go about adding features to your finished product.

This practical will involve you using the following Java classes: ServerSocket, Socket, String, BufferedReader, and BufferedOutputStream. Please take some time to look through the Java documentation provided for these classes. You can also find reasonable documentation of the HTTP protocol online at Wikipedia.

Academic Integrity

The files you submit must be your own work. You may discuss the assignment with others, but the actual code you submit must be your own and not taken from ChatGPT, Copilot or other LLM. You must fully also understand your code and be capable of reproducing and modifying it. If there is anything in your code that you don't understand, seek help until you do. Note that there will be code-related questions in the test and exam.

The due date for submitting this assignment on Moodle is **Friday, 30th of August,** and it is worth **10%** of your final grade. Besides submitting via Moodle, you need to have the code signed off at one of the labs (preferably before the deadline, but not later than two weeks after the submission deadline).

HTTP

Web servers listen on a socket, usually port 80, for unencrypted HTTP. Web servers and browsers use a protocol to retrieve files, which the browsers then display. The web browser initiates the request for a page. It opens a connection and sends a request message consisting of lines of text, each line separated by a *CR/LF pair*. The web browser signals to the web server that the request message is complete by sending an empty line.

Here is an example:

```
GET /study/ HTTP/1.1
Host: www.waikato.ac.nz
User-Agent: Mozilla/5.0 (X11; FreeBSD amd64; rv:78.0) Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Pragma: no-cache
Cache-Control: no-cache
```

The first line has a command verb (GET), the name of the resource (file) being requested (/study/) and the protocol version in use (HTTP/1.1). The other lines provide additional information about the web browser in use, the language requested if translations are available, and the web server specified in the browser's address bar (www.waikato.ac.nz).

The GET command is used to request a file, such as an HTML page, an image file, or another file. It is the most commonly used command and the only one you are required to implement. The web server responds on the same socket as used for the request with something that looks like this:

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Accept-Ranges: bytes
Age: 541190
Cache-Control: max-age=604800
Content-Type: text/html; charset=UTF-8
Date: Wed, 21 Jul 2021 22:21:38 GMT
Etag: "3147526947"
Expires: Wed, 28 Jul 2021 22:21:38 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (sec/974D)
Vary: Accept-Encoding
X-Cache: HIT
Content-Length: 648

<file contents go here>
```

As before, the first line is the most important. It contains the protocol version (HTTP/1.1), a status code (200) and a brief text message explaining the status code (OK). Then comes more information about the software running on the web server, as well as the file being sent back, including the date it was last modified, a unique ID, the number of bytes being sent back, and what it will do with the socket when the request is done. As with the request header, the response header is in plain text, and the end of the response header is signalled with an empty line.

The data that follows may not necessarily be a text file; it might be an image file, which is a binary file. Therefore, when your web server sends the file to the browser, it must be careful not to corrupt it by reading it a line at a time and storing it in a String, as the String will not retain all of the bytes read.

Our web server will use two status codes. 200 means that everything is OK and the file requested is going to be returned. 404 is used when the file does not exist on the web server. A 404 looks something like this:

```
HTTP/1.1 404 File Not Found

foo.txt not found
```

In this case, the web browser (or the line command called "wget", see below) uses the 404 code to know that the requested file did not exist on the server. The empty line signifies the end of the HTTP header. A web browser would display the string "foo.txt not found" to the user.

Step 1: The basic console application

The server will be a console application (no GUI). Start with a program that looks like the following. The file starts with all the classes necessary for this project imported. Add a line to write ``Web Server starting'' to the console.

```
import java.net.*;
import java.io.*;
```

```
import java.util.*;
class HttpServer
{
    public static void main(String args[])
    {
        //write something to the console here
    }
}
```

You need to choose a port for your web server's ServerSocket. In the lab machines, you should choose a port number larger than 50,000 but below 65535; you will not be able to use the default web server port number of 80 (you would need to be a super-user or root to do that). You can code the port number into your application or have it specified on the command line.

Testing: Ensure your program compiles and runs. It will not produce output, but it should be complete.

Step 2: Accept a connection

The next step is to accept a connection and then spawn a thread to process that connection. Declare a new class HttpServerSession, which extends Thread. The HttpServerSession constructor should take a single argument -- the socket that was accepted, and save it in a variable private to that class.

Take a look at ChatServer.java in the source code folder for an idea about this.

Testing: Add a statement to HttpServer::main() that prints a message when a connection is made. Then, open a web browser and enter **http://lab-rg06-21:50000/cat.jpg** in the location bar (substituting lab-rg06-21 for the name of the machine where you are running the HttpServer). Your HttpServer should print out a message to the console saying a connection was received, specifying which IP address (the client) made the connection.

Note: using a web browser *when you're not physically located in the Hamilton Linux lab* might be frustrating. One option is to use the **wget** tool from the console while logged into one of the Hamilton Linux lab systems, as in:

```
wget http://lab-rg06-21:50000/cat.jpg
Step 3: Look at the HTTP request
```

In the HttpServerSession::run method, declare a BufferedReader that is connected to the socket's InputStream. Get the request by using the readLine() method. You will call readLine() until you get an empty line "" (the end of the well-formed request) or readLine returns null (the client has no more to send). Eventually, you will switch your println out with a line that calls your HttpServerRequest::process() method (next step) to parse the request.

Now is a good time to add code to your HttpServerSession that closes the connected Socket when you have finished with it.

Testing: add a statement to your run method to print out the request line to the console. Run your web server and browser as before; you should see the request printed out to the console by your web server.

Step 4: Send back a "Hello World" response

This part is a little more complicated due to Java's PrintWriter::println behaving differently on various platforms and HTTP's requirement that each line have a CR and LF pair. Instead of using PrintWriter, we will use BufferedOutputStream. You can add a method to your HttpServerSession that will *mimic the println method* found in PrintWriter, but be compliant with what HTTP requires.

```
private boolean println(BufferedOutputStream bos, String s)
{
  String news = s + "\r\n";
  byte[] array = news.getBytes();
  try {
      bos.write(array, 0, array.length);
  } catch(IOException e) {
      return false;
  }
  return true;
}
```

As said earlier, you only need to send back a single line of status text. Try sending back a 200 message like the one in the earlier example; ensure you send an empty line -- println("") -- and then send the string "Hello World". You should see the words "Hello World" in your browser, or "Hello World" in the file that the command wget saved.

Test: run your web browser now. It should display "Hello World" in the browser. If you are using wget, look inside the output file saved.

Step 5: Parse the HTTP request

Create a new Java class (in its own file) for parsing and using the HTTP request. I suggest you call it HttpServerRequest.java, and I've provided you a skeleton to get you started, which you can download here.

You can obtain each of the three parts of a GET request string by using code like this:

```
String parts[] = in.split(" “);
```

A well-formed GET request has exactly three parts in the array (using parts.length).

The first part should be the "GET" string; ensure that is the case by using parts[0].compareTo("GET"); this method returns zero if parts[0] is the "GET" string. The next entry in the array is the file being requested. You can remove the first slash in the request with something like

```
String filename = parts[1].substring(1);
```

so that your HttpServer returns files relative to the directory in which you are running it. This use of the substring() method returns everything after the 1st character in the string. If you wanted to extract everything after the 4th character, you'd pass 4 to substring().

You also need to extract the contents of the Host: header, if the client supplied it. You can use the startsWith() String method to determine if the line starts with "Host: " and then use substring() to get the text that follows "Host: ".

If the filename ends with a "/" your class must append index.html to the filename. You can use the endsWith() String method to determine that.

The skeleton supplied has a "line" variable that I suggest you increment each time HttpServerRequest::process() is called. The first line has the GET line, all other lines could have optional headers, and an empty string ("") or a null String parameter means end of header.

Step 6: Join HttpServerRequest with HttpServer

Earlier, you sent back a response with ``Hello World'' where the actual contents of the file should go, regardless of what the browser requested. The goal of this step is to join HttpServerRequest with HttpServer. To start with, replace the loop that reads until readLine() returns an empty string with a loop that continues until HttpServerRequest::isDone() returns true. Inside that loop, call readLine() and pass the String that readLine() returns to HttpServerRequest::process(). You should now be able to call HttpServerRequest::getFile() and HttpServerRequest::getHost() once you are out of the isDone() loop to get the File and Host, if they exist.

Step 7: Return the requested file

The goal of this step is to return the actual contents of the file. To do this, you're going to have to declare a byte array of a fixed size, open the file with a FileInputStream, read from the file with the FileInputStream::read method, and with each read, send the byte array to the client using the BufferedOutputStream::write method. Be sure to catch the end of the file (when FileInputStream::read returns -1). At the end of the file, ensure that you then flush() the output before you return.

Your assignment needs to support what we call "Virtual Hosting": the web server hosts multiple websites. This is accomplished using the **Host:** header in the request. You will do this by creating two directories that match the domain name and port combinations that your web browser sends. For example:

```
localhost:50000
lab-rg06-17:50000 (if you are running your web server on lab-rg06-17)
```

Place different html and image files inside of each. If the getHost() method returns null, then return files as if the user had entered "localhost:50000". To get the path, it is simply a case of concatenating the Strings as follows:

```
host + "/" + file
```

Test: You should have a working web server. Test that you do by putting a web page in your directory, and also test image files.

The best way to test image files is to use wget, as follows:

```
wget http://lab-rg06-27:50000/image.png
md5sum output.file path/to/image.png
```

where path/to/image.png is the location of image.png in the local file system. If the files are identical, they will have the exact same md5sum. If they do not have the exact same md5sum, you have a bug in your code. Consider looking at the size of the files for a clue on what might be going wrong, as in:

```
ls -l output.file path/to/image.png
```

Step 8: Deal with missing files

You may have noticed your web server throwing an exception when your browser automatically requests favicon.ico. Extend your web server to catch that exception and return the 404 message detailed earlier. Extend your web server to log a message that says whether or not the requested file was found or not.

Test: Check that your web server sends a 404 message by requesting a file that you know does not exist.

Step 9: Running over a slow line

People sometimes use web browsers over slow connections. It is important that a web server can service all requests simultaneously, rather than be held up servicing a slow client. This is achieved by threading your web server, such as the ChatServer example (see Lecture and code shared).

It is difficult to provide a slow environment in the lab, though you can modify your HttpServer to make it go slow when sending a file back. To check this, please add a sleep(1000) to the loop where you read the file and send it to the client in pieces. This will increase the time it takes for a large file to be sent back.

Test: place some reasonably large (50K) image files in your directory. When your browser fetches an image file, you should be able to see the browser gradually render the image as each chunk is delivered to it. If you do not see the effect, try a different image file, or create a large text file with lines repeated over and over. You should see the browser's scroll bar gradually provide a greater scrolling range as each chunk arrives.

Once you have observed your web server behaving correctly (able to service multiple clients simultaneously), comment out the sleep line.

Grading

Your solution will be marked on the basis of how well it satisfies the specification, how well-formatted and easy to read your code is, and whether each block of code has at least some comment explaining what it does and why.

Your code should compile and run as a console program from the Linux command line (i.e. no GUI). Students are encouraged to compile and test their code on the Linux machines in R Block prior to submitting their solutions.

Make sure your name and student ID number are included in the header documentation of each source code file. You may include a readme.txt file with any notes you wish the marker to read as they are marking your submission.

If you have the assignment verified in the lab prior to the due date, your feedback will be supplied verbally, and you will receive a grade after you have submitted your assignment.

The grading rubric is as follows:

- 5 marks: HttpServer accepts well-formed requests from clients, and delivers requested resources without introducing corruption. HttpServer returns a well-formed 404 response when the client requests a resource that the server does not have
- 2 mark: HttpServer implements virtual hosting (server hosts multiple websites)
- 2 mark: submission deals properly with malformed requests
- 1 mark: HttpServer and HttpServerRequest is well commented.