CS5001 Object-Oriented Modelling, Design and Programming
Practical 3 – Networking

School of Computer Science
University of St Andrews

Due Friday week 8, weighting 20%
MMS is the definitive source for deadlines and weightings.

In this practical, you are required to write a simple Java web server which can respond to some HTTP/1.1 requests [1].

Setup

For this practical, you may develop your code in any IDE or editor of your choice. However, please ensure that you create a suitable assignment directory `CS5001-p3-networking` on the computer you're using. All your source code should be in a `src` directory inside your assignment directory, `CS5001-p3-networking/src` and your main method should be in a `WebServerMain` class in `WebServerMain.java`.

You should probably copy the www directory from StudRes (containing some sample html pages and images) to your assignment directory. This path can be found on the host machines, which you can access remotely in the usual way. You can download the whole directory using `sftp` and the `get -r` command. You can then add more files to your local `www` directory when testing your server if you want to.

Basic Requirements

At a minimum your webserver program should satisfy the following basic requirements.

- The main method for your server should take two command-line arguments: the directory from which your server will serve documents to clients; and the port on which your server should listen.
- The server should support and respond correctly to `HEAD` requests.
- The server should support and respond correctly to `GET` requests.
- The server must be able to return HTML documents requested by a client.
- The server should respond with appropriate error messages when non-existent services or resources are requested.

Advanced Requirements

For the highest grades, your program should also support one or more of the following advanced requirements.

- Returning of binary images (GIF, JPEG and PNG).
- Multithreading – support multiple concurrent client connection requests up to a specified limit.
- Logging – each time requests are made, log them to a file, indicating date/time request type, response code etc.
- Supporting other methods in addition to `GET` and `HEAD`.

Implementation

You must use sockets when writing your own server for this practical and should not rely on existing HTTP library code from Oracle or other third parties. The server should be able to serve documents from a specified document root and listen on a specified port. When it receives an incoming request, the server will have to read text over the socket from the client and check whether the textual request corresponds to a `HEAD` or `GET` request. If it does, it should try to serve the requested response header for a `HEAD` request, and for a `GET` request, the response header plus body (the latter containing the requested file data) from its document root over the opened socket connection to the client (browser).

More specifically, your server should listen for incoming connection requests. When a client connects and sends a request to the server, it should examine the text of the request. If your server does not support the received request, it should send back an appropriate response header (as indicated in lectures).

For `HEAD` requests, your server should send back an appropriate response header containing the information about the resource identified in the request (if the file exists at the specified location in the document root). For a `GET` request, your server should similarly send back an appropriate response header (as for the `HEAD` request), followed by the content of the requested file (assuming the file exists at the specified location in the document root). In either case, if the requested file cannot be found, your server should send back an appropriate response header, potentially followed by some error page (the latter for a `GET` request only).

Note that both response header and content must be formatted as indicated in lectures or in [1], and must be sent back over the socket connection to the web browser, not merely printed to `System.out`. That said, you may wish to also include `System.out.println` messages to help you debug your server as you develop it.

Once your server has responded, it should flush and close the connection to the client and listen for further requests. That is, your server is not required to keep connections alive.

Compiling and Running

For your program to be compatible with the automated checker:

- It must be possible to compile all your source code from within the `src` directory using the simple command:

```
javac *.java
```

- It must be possible to get your program to serve pages from a directory and port specified on the command line that are passed to your `main` method in the `String[] args`. For example, executing the following command from within the `src` directory should permit your program to serve pages from the `CS5001-p3-networking/www` directory (i.e. one level up from the `src` directory) and listen on port 12345:

  ```
  java WebServerMain ../www 12345
  ```

- If your server is started without supplying the command-line arguments, it should simply print the usage message indicated below and exit:

  ```
  Usage: java WebServerMain <document_root> <port>
  ```

Running the Automated Checker

As for previous practicals, we have provided some basic tests. In order to run the automated checker on your program, upload your solution to a directory on the School server, log into the lab machines through SSH, **change directory** to your `CS5001-p3-networking` directory, and execute the following command:

```
stacscheck /cs/studres/CS5001/Practicals/p3-networking/Tests
```

If you use a Linux or Mac machine, you can also install stacscheck on your own computer. When you are ready to submit, make sure that you also run the checker on the archive you are preparing to submit, by calling, for example:

```
stacscheck --archive p3.zip /cs/studres/CS5001/Practicals/p3-networking/Tests
```

As in previous practicals, if the automated checker doesn't run, or the build fails, or all comparison tests fail, you may have mistyped a command or not have followed the instructions above.

Tests 1–14 check basic operation of your server. The first comparison test checks whether the correct usage message is displayed when the command-line arguments are not supplied to your server. The remaining tests check response header and content for a number of `GET` and `HEAD` requests and one unsupported request. If one or more of these fails, then you are possibly not sending back the expected header or content in your response where appropriate. In case of failures, the tests should indicate what your program has sent back over the socket and what we expected. Possible areas to examine include:

- opening, sending data on, flushing, and closing connections properly;
- use of `<CR>` and `<LF>` to delimit header fields and header from content (as indicated in lectures);
- use of appropriate header fields and values in the response header;
- sending back the correct file content for the given request.

The final test runs the style checker over your source code.

Test and Debug

In order to test your web server program manually, you should be able to use any browser on the local machine and try to access http://localhost:12345/index.html to access index.html from your server listening on port 12345 on the same machine.

Take care to put sufficient debug statements (using e.g. System.out.println) into your code so as to help you locate errors in your code and e.g. print out the incoming request strings on the server.

If you run into problems, you may find it useful to use simple client programs to send simple requests to your server and print out the reply. You could use Curl & Telnet (on Linux) or Curl & Putty in raw telnet mode (on Windows – see https://curl.haxx.se/download.html & https://www.putty.org/) as simple client programs to test your server. On Linux, you could for example type

```
curl -s -I -X GET localhost:12345/index.html
curl -s localhost:12345/index.html
```

to send a `GET` request for index.html to a server listening on port 12345 and display the response header and body respectively. You can send a `HEAD` request by replacing `GET` with `HEAD` in the first curl command above. You could also for example type

```
telnet localhost 12345
```

in a Terminal window to connect to a server you have listening on port 12345 on the same machine. If your server program accepts the connection, you can then manually type in a request into Telnet (such as `GET /index.html HTTP/1.1`) and check the responses from your server program and its debug output.

In your own code, take care to flush sockets and close them cleanly or else you may not see the responses made by the server on the client and you may have issues running your server program due to sockets being left open.

You might also consider performing more rigorous automated testing. If you decide to adopt this approach, it is probably a good idea to start by looking at the tests we have made available to you at `/cs/studres/CS5001/Practicals/p3-networking/Tests`. You can create new tests in a local sub-directory in your assignment directory and pass the directory of your own tests to stacscheck when you run it from your assignment directory. Also, you should look at the documentation for the automated checker at

https://studres.cs.st-andrews.ac.uk/Library/stacscheck/

Deliverables – Software (and Readme)

Along with your source code, you must include a short readme file which lists any of the *advanced requirements* you have implemented in your assignment directory (`CS5001-p3-networking`). You should hand in a zip archive of your assignment directory (containing all your source code, any sub-directories, and readme), via MMS as usual.

You do not need to include a readme if you have not attempted any advanced requirements. However, you should include a readme if there is anything else in your solution that you want to draw our attention to.

Marking

A very good attempt at satisfying the basic requirements above **in an object-oriented fashion** can achieve a mark of 14–16. This means you should produce very good, re-usable code which makes proper use of inheritance, association, and encapsulation with very good method decomposition. To achieve a 17 or above, your code must in addition make a very good attempt at some of the advanced requirements. Take care to test your solution exhaustively. See the standard mark descriptors in the School Student Handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8-hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/students/rules/academicpractice/

References

[1]: Berners-Lee, T., Fielding, R., Frystyk, H. et al. (1999), "Hypertext Transfer Protocol – HTTP/1.1", RFC 2616, available from: https://www.w3.org/Protocols/rfc2616/rfc2616.html