University of
St Andrews

School of Computer Science

CS5001

Practical 3 – Networking

Parts completed:

- All basic requirements
- Returning of binary images through GET requests
- Multithreading support for multiple clients to a given limit
- Logging of connections made by clients, their requests, and the response codes returned
- Supported additional commands. However, not web commands. Commands to terminate the client connection and terminate the server were implemented.

ID: 170018405

Table of Contents

1

1.0 Program requirements 2

2.0 Implementation 2

2.1 WebServerMain 2

2.2 WebServer 2

2.3 ConnectionHandler 3

2.4 RequestHandler 3

3.0 Testing 4

4.0 Running instructions 5

1.0 Program requirements

In this practical a basic Java web server was required. The basic requirements of the webserver included to support HEAD and GET requests – returning full HTML documents when required, or, the appropriate error messages when a resource cannot be found or a method is not implemented. In addition to the above basic requirements, a decision was made to include support for:

- returning binary images upon a GET request
- multiple clients concurrently up to a specified limit
- client request (and server response) logging in a text file

2.0 Implementation

To implement all of the above, a decomposition of functionalities was carried out. Following loosely from the example shown in the course material, the following classes were built.

2.1 WebServerMain

The purpose of this class is to provide a main method to launch the web server. The class is also responsible for ensuring that valid command line arguments are provided.

To check the validity of the command line arguments, extensive checks are made before a WebServer object can be created. These checks include checking the number of command line arguments, as well as if those arguments are valid.

For example, the directory path provided in the first command line argument is checked to actually represent a valid directory. Moreover, the port number given as a second command line argument is checked if it can be parsed as a number and if the number is between the values of 0 and 65535.

2.2 WebServer

This class as suggested by its name, handles all the "server" specific tasks. This means acting almost as a router – allocating clients to their own connection handlers. The server achieves this functionality through the launch method. Given a new connection request, the web server allocates the client with its own connection handler on one of the existing threads (from a thread pool), and continues listening for new connections.

The class provides three separate constructors allowing for flexibility and choice when making WebServer Objects. The first constructor only requires two parameters, a root directory to serve resources from, and a port to listen for connections on. The second constructor adds another parameter which gives users the option to turn verbose mode on. By default, this mode is off if the first constructor is used. The last constructor allows users to specify if they would like the webserver to log the connection transactions with the clients on a text file. Nonetheless, the functionalities offered by the latter two constructors can be still accessed by using the setter methods provided to turn the verbose and logging modes on.

Note: if the logging mode is used, the WebServer will write to a file which it will create in a directory named logs (one directory up from the src directory). If this filename and directory need to be changed, the setOutFile method should be used to provide a new path and file name to the server.

For example, to create a file called logs in a directory called connections, the following argument can be given to the method:

setOutFile("../connections/logs.txt");

It is at this point that the Config class should be mentioned. This class contains all the constants that may be changed by a user without editing the code to change the server configurations. The THREADS field in particular is of importance as through this, the number of threads supported by any WebServer object can be changed. In a future implementation, this way of setting the number of threads would be changed to avoid locking all WebServer objects to the same number of threads. One way to do this, is by providing a none-static setter method to set the number of threads of every WebServer object independently.

2.3 ConnectionHandler

The connection handler class was made to manage client connections concurrently while the WebServer continues to listen for new connection requests. A connection handler's job is thus to act as a gate keeper, managing the flow of data in, and passing data through the socket to the client. This class thus also handles the connection termination and timeout logic, as well as the logging functionalities.

Two constructors are provided depending if users want the ConnectionHandler to log requests and responses from a client. The only difference between the two constructors is an extra parameter named outFile. This is the file path and name to log the connection information on.

You may also notice a ServerActive Object is passed as the last parameter. This is a bespoke object created by the WebServer to help decide if it should continue listening for new incoming connections. Given an appropriate command from a client, the ConnectionHandler can use this object to stop the WebServer from listening for new connections, and terminate once all the open connections have been terminated.

To allow for both binary (e.g. jpg) and character based output (like HTML headers) to be written over the socket, the class utilises both a BufferedOutputStream and a PrintWriter respectively. As for reading the client input, a BufferedReader is used in conjunction with an InputStreamReader.

This class implements the runnable interface which allows for threads to be used. The class's main logic is thus executed within the run method.

In the run method, a while loop is used to handle the client requests until the client either terminates the connection, or a connection timeout occurs. Note: the timeout duration is set in the Config class.

To avoid including all the logic required to deal with the client requests within the run method (which would essentially make the ConnectionHandler class none-reusable), a separate class was implemented to contain the code for decoding and responding to the client requests. By doing this, the types of supported HTTP methods and their corresponding responses can be altered without needing to alter either the ConnectionHandler or the WebServer classes.

2.4 RequestHandler

Arguably the main challenge associated with writing the Java WebServer program so that it is both reusable and extendable, came from finding a way to provide some sort of extendable interface by which users of the WebServer/ClientHandler classes can just write the code for the methods they want to support. Several ideas were explored. For example, a Commands class which utilises reflection to call a method that matches a client request was considered. However, it was determined that this approach is both too slow, doesn't really follow good object oriented practices, and doesn't provide a specification for what parameters and return types each method should have.

The solution taken in the end was to create an abstract class named RequestHandler to specify the command that should be implemented in a subclass to male it compatible with the ConnectionHandler class.

The method required to be implemented is the respondToRequest method, which is actually called by both the setNewRequest and one of the constructors provided by the abstract class. In this method, users are instructed to write the logic for responding to client requests. The SimpleRequestHandler class is an example subclass of RequestHandler which shows a possible way to implement all the HTTP methods that are supported.

As it can be seen, if there is a need to support more methods, all that would be required is to create a new class which extends SimpleRequestHandler. Once extended, the respondToRequest method can then be overridden, and additional methods can be added.

Like the WebServer and ConnectionHandler classes, the RequestHandler classes provide flexibility of use, making them useful for many use cases. One way to use the class is to create a new instance of it (using the secondary constructor) every time a new request is received. Since the secondary constructor automatically calls the setNewRequest and thus the respondToRequest methods, no additional methods need to be called to elicit a response. Alternatively, the primary constructor could be used to create a single RequestHandler object, which is used numerous times to respond to new requests via calls to the setNewRequest method. Not only this, but the user may also use the public response methods (such as the "head" or "get" methods) directly by passing a file object (containing the resource to send to a client) to each as a parameter. This last usability feature allows users of the class to send resources to a client without the client necessarily making a request. Hence, the setNewRequest method can be bypassed.

3.0 Testing

Testing was carried out for each class separately during the programming phase (ensuring all methods perform the functionalities expected). In this section though, the testing of the system as a whole is discussed.

To test the system, web browser and telnet clients were used.

Things to test for included:

- HEAD requests return the appropriate content response code, MIME types and, file sizes (lengths).
- HEAD requests of invalid sources returned the header information of the 404 html page written within the resources directory.
- HEAD requests where no specific resource path is given returned the home (index.html) page.
- Similarly, GET requests for valid, invalid, and no resource returned the expected results. In the case of invalid resource paths, the 404 html page should be returned.
- Since a binary output stream is used to return content when a GET request is made. A web browser was used to check pictures are correctly sent by the ConnectionHandler to the client. For example, the following link is used to check the output of a GET request for a jpg file in the resources directory: http://localhost:12345/tp_it.jpg
- Similarly, a web browser is used to check that the server returns the correct webpages (displayed as intended) by navigating using the links provided on the web pages.
- To test for the above, the following link is used to make a connection to the server: http://localhost:12345/index.html. Then, the links on the pages are clicked to move between the other pages.

- Unsupported requests were then checked. The RequestHandler is coded in a way to ignore empty requests, and this can be checked by sending entering returns through a telnet client.
- Similarly, requests that do not follow the HTTP/1.1 protocol (<method> <resource> <protocol>), were entered to check the responses the server makes. The server is coded to respond to any request as long as the first part of the request (the method) is supported. However, when unsupported methods are given, the server will return a 501 response (containing the header information of the 501 html page in the resources directory).
- The timeout duration was tested by making a connection to the server using a telnet client and timing how long it took for the server to terminate the connection given no requests were made.
- Multithreading was tested by reducing the number of threads supported in the Config class to 2, and connecting 3 or more telnet clients to the server, then checking how many clients the server responded to. Once it was ensured that the server responded to only 2 clients. One client at a time was terminated, and it checked if the server began to respond to the clients that were put on the waiting list.
- While testing the above, the logging feature was also tested, making sure every client was registered as connected, and their requests/responses well all recorded.
- The exit command (which can be sent to the server as "!exit" through a telnet client) causes the server to terminate the connection with the client and free up a thread.
- The server exit command (sent "!serverexit") causes the server to stop listening for new requests, and terminates the program once the last client exits.

4.0 Running instructions

It is recommended that the resources directory provided in the submission folder be used with the source code provided. This is because the source code makes use of the 404.html and 501.html pages to return 404 and 501 responses. Furthermore, a logs folder should be provided one directory up from the src directory, as the code looks for this directory to create log files in.

To run the Web server application, the following command should be executed from the terminal within the src directory:

$$java\ WebServerMain\ ../resources/www\ <port>$$

The code can be compiled with the following command:

$$javac\ *.java$$