CIS 457: Data Communications
Week 05 Assignment: Building a Multi-Threaded Web Server
Point Value: 10 Pts, late submission policy (20%/day for up to 5 days)
Due Date: 2/16/2024 by 12:00 AM EST
Submission format:

The following reports will be submitted on Blackboard (BB) using each group member's account.
(1) The entire assignment's solution in a PDF format (your contribution to the assignment)
(2) Group's Agreed-upon solution in a PDF format.
(3) Group Code as requested in the assignment.
Note: Missing an individual's contribution leads to receiving **NO** credit for the assignment.

Objectives

The purpose of this lab is to:

- Identify the difference between a single-thread and multi-threaded application.
- Develop a Multi-Threaded Web server in two steps.

Note:

- Guided instructions for this assignment are given in Java programming language. However, feel free to ignore these guided steps, write and debug your own code in any language you wish.

Section 1: Identify the difference between single thread and multithreaded application (1 Pt)

Copy the Echo Server app code into your home directory. Then, familiarize yourself with the application code and compile it. If you do not remember how to compile and run your app, check this out in the TCP/UDP Sockets programming assignment:

1. Run the single thread version of the app using 2 clients simultaneously and then answer the following question:
Question 1: Will the server be able to service the two clients simultaneously? and why?

2. Run the multi-threaded version of the same app using 2 clients simultaneously and answer the following question:
Question 2: Will the server be able to service the two clients simultaneously? and why?

Section 2: Develop a Multi-Threaded Web Server in Two Steps (9 Pts)

In this section, you will have built a multi-threaded Web server capable of *processing simultaneous service requests in parallel*. You should be able to demonstrate that your Web server can deliver the attached index.html file to a Web browser. You may read about multi-threading here: https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

We will implement a multi-threaded web server, as defined in RFC 1945. In the *main thread*, the server listens on a fixed port. When it receives a TCP connection request, it sets up a TCP connection through another port and services the request in a separate thread. To simplify this programming task, we will develop the code **in two stages**:

- In ***the first stage***, you will write a multi-threaded server that simply *displays the* ***contents*** of the HTTP request message it receives.
- In ***the second stage***, you will add the code required to generate an appropriate ***response message***.

As you develop the code, you can test your server from a Web browser. But remember that you are not serving through the standard port 80, so you need to specify the port number within the URL you give to your browser. For example, if your server is listening on port 6789, and you want to retrieve the file *index.html*, then you would specify the following URL within the browser:

http://127.0.0.1:6789/index.html

If you omit ":6789", the browser will assume port 80, which most likely will not have a server listening on it.

Stage 1: Web Server in Java- Part A (6 Pts)

In the following steps, we will go through the code for the first implementation of our Web Server. Wherever you see **"?"**, you will need to supply a missing detail in the given code with this lab assignment.

Our first implementation of the Web server will be multi-threaded, where the processing of each incoming request will take place inside a separate thread of execution. This allows the server to service multiple clients in parallel or to perform multiple file transfers to a single client in parallel. When we create a new thread of execution, we need to ***pass to the Thread's constructor an instance of some class that implements the Runnable interface***. This is the reason that we define a separate class called **HttpRequest**.
So, the code for the application in this stage will have:

- **2 files** (1) WebServer.java and (2) HttpRequest.java.
- Also, place the *index.html* file in the same folder where the application code resides.
- These files are explained in the following section. Please use the following explanation along with the given Java files to supply the missing code:

➔ Explanation of the WebServer.java:

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

public final class WebServer
{
        public static void main(String argv[]) throws Exception
        {
                . . .
        }
}
……..
……..
```

Normally, Web servers process service requests that they receive through well-known port number 80. You can choose any port higher than 1024, but remember to use the same port number when making requests to your Web server from your browser (in this app we will use port **6789**).

```
public static void main(String argv[]) throws Exception {
        // Set the port number.
        int port = 6789;
        int port = (new Integer(argv[0])).intValue();
        . . .
}
```

Next, we open a socket and wait for a TCP connection request. Because we will be servicing request messages indefinitely, we place the listen operation inside an infinite loop. This means that we will have to terminate the Web server by pressing ^C on the keyboard.

```
// Establish the listen socket.
                ?
// Process HTTP service requests in an infinite loop.
while (true) {
        // Listen for a TCP connection request.
        Socket connection =?;
. . .
}
```

When a connection request is received, we create an ***HttpRequest*** object, passing to its constructor a ***reference to the Socket object that represents our established connection with the client***. In order to have the **HttpRequest object** handle the **incoming HTTP request** in a separate thread, we first create a new Thread object, passing to its constructor a reference to the ***HttpRequest*** object, and then call the thread's **start ()** method.

Please, note that after the new thread has been created and started, execution in the WebServer's main thread returns to the top of the message processing loop. The main thread will then block, waiting for another TCP connection request, while the new thread continues running. When another TCP connection request is received, the main thread goes through the same process of thread creation regardless of whether the previous thread has finished execution or is still running. This completes the code in main ().

```
// Construct an object to process the HTTP request message.
HttpRequest request = new HttpRequest(?);

// Create a new thread to process the request.
Thread thread = new Thread(?);
// Start the thread.
?;
 }}}
```

********** **End of WebServer.java Code**********

For the remainder of the assignment, it remains to develop the **HttpRequest class** as follows:

➔ **Explanation of the HttpRequest.java:**

We declare two variables for the HttpRequest class: CRLF and socket. According to the HTTP specification, we need to terminate each line of the server's response message with a carriage return (CR) and a line feed (LF), so we have defined *CRLF* as a convenience. The variable *socket* will be used to store a reference to the connection socket, which is passed to the constructor of this class. So, the structure of the ***HttpRequest*** *class* is shown below:

```
final class HttpRequest implements Runnable
{
        final static String CRLF = "\r\n";
        Socket socket;

        // Constructor
        public HttpRequest(Socket socket) throws Exception
        {
                this.socket = socket;
        }

      // Implement the run() method of the Runnable interface.
      public void run()
        {
                . . . //Provide implementation for the method run () such as explained below
        }

        private void processRequest() throws Exception
        {
                . . . //Provide implementation for the method processRequest() such as explained below

        }
}
```

In order to pass an instance of the **HttpRequest** class to the Thread's constructor, *HttpRequest must implement the Runnable interface*, which simply means that we must define a public method called run() that returns void. Most of the processing will take place within processRequest(), which is called from within run().

Up until this point, we have been throwing exceptions, rather than catching them. However, we cannot throw exceptions from run(), because we must strictly adhere to the declaration of run() in the Runnable interface, which does not throw any exceptions. We will place all the processing code in processRequest(), and from there, throw exceptions to run(). Within run(), we explicitly catch and handle exceptions with a try/catch block. **// Implement the run() method of the Runnable interface.**

```
public void run()
{
```

```
        try {
                processRequest();
        } catch (Exception e) {
                System.out.println(e);
        }
}
```

Now, let's develop the code within processRequest(). We first obtain references to the socket's input and output streams. Then we wrap InputStreamReader and BufferedReader filters around the base input stream. However, we won't wrap any filters around the output stream because we will be writing bytes directly into the output stream.

```
private void processRequest() throws Exception {
        // Set up an output stream.
                DataOutputStream os = new DataOutputStream(?);
        // Set up an input stream.
                BufferedReader br = new BufferedReader(new InputStreamReader(?));
        . . .
}
```

Now, we are prepared to get the client's request message, which we do by reading from the socket's input stream. The readLine() method of the BufferedReader class will extract characters from the input stream until it reaches an end-of-line character, or, in our case, the end-of-line character sequence CRLF.

```
The first item available in the input stream will be the HTTP request line.
//Read the request line of the HTTP request message.
String requestLine = ?;

    // Display the request line.
    System.out.println();
    System.out.println(requestLine);
```

After obtaining the request line of the message header, we obtain the header lines. Since we don't know ahead of time how many header lines the client will send, we must get these lines within a looping operation.

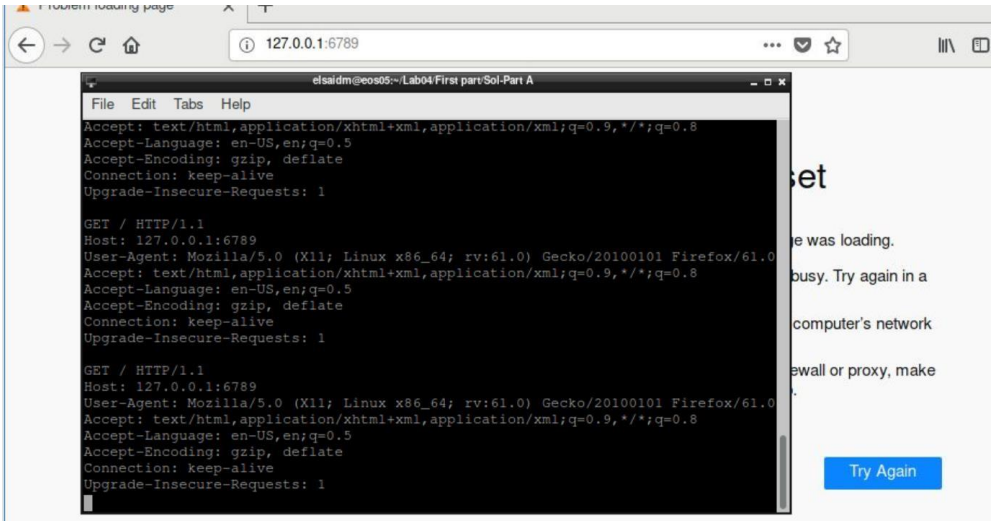
```
// Get and print the header lines.
String headerLine = null;
while ((headerLine = br.readLine()).length() ?) {
  System.out.println(?);
}
```

We don't need the header lines other than to print them to the screen, so we use a temporary String variable, headerLine, to hold a reference to their values. The loop terminates when the expression (headerLine = br.readLine()).length() evaluates to zero, which will occur when headerLine has zero length. This will happen when the empty line terminating the header lines is read.

Add the following lines of code to close the streams and socket connection.

```
// Close streams and socket.
os.close();
br.close();
socket.close();
```

In the next step of this assignment, we will add code to analyze the client's request message and send a response. But before we do this, ***let's try compiling our program and testing it with a browser to direct a web request to our server.*** After your program successfully compiles, run it with an available port number and try contacting it from a browser. To do this, you should enter into the browser's address text box the local IP address of your running server, such as http://127.0.0.1:6789/ The server should display the contents of the HTTP request message in **your terminal window as shown below**.



Based on your program outcome, answer the following question:

Question 1

a) Does the format of the obtained http request adhere to the HTTP protocol specification?
b) Provide a screenshot of the program output from your terminal and a copy of your code. (add a copy of your code with your own individual submission).

Stage 2: Web Server in Java returning a response back to the client - Part B (12 Pts)

Instead of simply terminating the thread after displaying the browser's HTTP request message, we will analyze the request and send an appropriate response. We will ignore the information in the header lines and use only the file name contained in the request line. In fact, we will assume that the request line always specifies the GET method and ignore the fact that the client may be sending some other type of request, such as HEAD or POST.

We extract the file name from the request line with the aid of the **StringTokenizer** class. First, we create a StringTokenizer object that contains the string of characters from the request line. Second, we skip over the method specification, which we have assumed to be "GET". Third, we extract the file name. As a guide, you may use the PowerPoint slides (**Chapter 02-Part III** - posted on Blackboard) for how to build a web server. This process would require making changes in the file **HttpRequest.java** and a minor change in the **WebServer.java**. For changes to be made in the **HttpRequest.java**, please use the following code and supply any missing pieces.

// Extract the filename from the request line.
? tokens = new StringTokenizer**(?);**

tokens.nextToken(); // skip over the method, which should be "GET"
String fileName = tokens.nextToken();

// Prepend a "." so that file request is within the current directory.

?

Because the browser precedes the filename with a slash, we prefix a dot so that the resulting pathname starts within the current directory.

Now that we have the file name, we can open the file as the first step in sending it to the client. If the file does not exist, the FileInputStream() constructor will throw the FileNotFoundException. Instead of throwing this possible exception and terminating the thread, we will use a try/catch construction to set the boolean variable fileExists to false. Later in the code, we will use this flag to construct an error response message rather than try to send a nonexistent file.

```
// Open the requested file.
FileInputStream fis = null;
boolean fileExists = true;
try {
        fis=?;

} catch (FileNotFoundException e) {
fileExists = false;
}
```

There are three parts to the response message: *the status line, the response headers, and the entity body*. The status line and response headers are terminated by the character sequence CRLF. We will respond with a *status line*, which we store in the variable statusLine, and a *single response header*, which we store in the variable contentTypeLine. In the case of a request for a nonexistent file, we return *404 Not Found* in the status line of the response message and include an error message in the form of an HTML document in the entity body.

```
// Construct the response message.
String statusLine = null;
String contentTypeLine = null;
String entityBody = null; if
(fileExists) {
        statusLine = ?;
        contentTypeLine = "Content-type: " +
                contentType( fileName ) + CRLF;
} else {
        statusLine = ?;
        contentTypeLine = ?;
entityBody = "<HTML>" +
                "<HEAD><TITLE>Not Found</TITLE></HEAD>" +
                "<BODY>Not Found</BODY></HTML>";
}
```

When the file exists, we need to determine the file's MIME type and send the appropriate MIME-type specifier. We make this determination in a separate private method called contentType(), which returns a string that we can include in the content type line that we are constructing.

Now we can send the status line and our single header line to the browser by writing into the socket's output stream.

```
// Send the status line.
os.writeBytes(?);
 // Send the content type line.
os.writeBytes(?);
 // Send a blank line to indicate the end of the header lines.
os.writeBytes(?);
```

Now that the status line and header line with delimiting CRLF have been placed into the output stream on their way to the browser, it is time to do the same with the entity-body. If the requested file exists, we call a separate method to send the file. If the requested file does not exist, we send the HTML-encoded error message we have prepared.

```
// Send the entity body.
if (fileExists)       {
sendBytes(fis, os);
fis.close();

} else {
        os.writeBytes(?);
```

After sending the entity body, the work in this thread has finished, so we close the streams and socket before terminating.
We still need to code the two methods that we have referenced in the above code, namely, the method that determines the MIME type, contentType(), and the method that writes the requested file onto the socket's output stream. Let's first take a look at the code for sending the file to the client.

```
private static void sendBytes(FileInputStream fis, OutputStream os)  throws
Exception
{
  // Construct a 1K buffer to hold bytes on their way to the socket.    byte[]
buffer = new byte[1024];
  int bytes = 0;

  // Copy requested file into the socket's output stream.
while((bytes = fis.read(buffer)) != -1 ) {
 os.write(?, ?, ?);

  }
}
```

Both read() and write() throw exceptions. Instead of catching these exceptions and handling them in our code, we throw them to be handled by the calling method.

The variable, buffer, is our intermediate storage space for bytes on their way from the file to the output stream. When we read the bytes from the FileInputStream, we check to see if read() returns minus one, indicating that the end of the file has been reached. If the end of the file has not been reached, read() returns the number of bytes that have been placed into buffer. We use the write() method of the OutputStream class to place these bytes into the output stream, passing to it the name of the byte array, buffer, the starting point in the array, 0, and the number of bytes in the array to write, bytes.

The final piece of code needed to complete the Web server is a method that will examine the extension of a file name and return a string that represents it's MIME type. If the file extension is unknown, we return the type application/octet-stream.

```
  private static String contentType(String fileName) {

if(fileName.endsWith(".htm") || fileName.endsWith(".html")) {

return "text/html";

            }

            ? {

               return "image/gif";

            }

            if(fileName.endsWith(".ram") || fileName.endsWith(".ra")) {

return "audio/x-pn-realaudio";

            }
            return "application/octet-stream" ;

   }
}
```

This completes the code for the second phase of development of your Web server. Now, place the index.html file in the same folder where the application code resides and run the server. Open a browser and send a web request to the server running @ http://127.0.0.1:6789/index.html

(Assuming that your server will still listen on port number 6789. When you connect to the running web server, examine the GET message requests that the web server receives from the browser.

Question 2

a) What are the benefits of writing data to a buffer instead of directly to a stream?
b) Provide a screenshot of the program output from your terminal and the web browser. The screenshot from the browser should show the URL address bar you used for testing.
c) Submit a copy of the application code.

Question 3

a) Use the following link as a guide (or feel free to conduct your own research) to describe all the methods that can be used to implement a solution for web system performance tuning and optimization.
Here is the link: https://www.datadoghq.com/blog/monitoring-apache-web-server-performance/

Acknowledgments

The ideas in this lab are partially taken from Computer Networking: A Top-Down Approach Featuring the Internet by Kurose and Ross Addison Wesley.