CS4065 Computer Networks and Networked Computing

Programming Assignment 1

This programming assignment is to be completed by each student in the class (i.e., **individual** work). No group work or collaboration is allowed. In this assignment, you will be tasked with two main tasks:

Task1 – Implement a multi-threaded web server in ***Java*** or ***Python***. A ***tutorial*** on socket programming in Java that can help you with this assignment and a future assignment is posted on canvas under the assignment module. If you choose to write code in Python, please check the last section of chapter 2 of the textbook (8th edition), for a discussion on socket programming in Python.

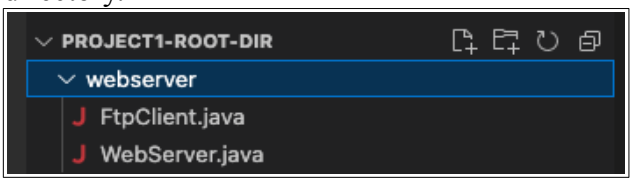
Task2 – Extend your web server to act as an FTP client if a user requested a text file (.txt) from within the browser. The FTP client will be responsible for connecting to a local FTP server set up by you and downloading the requested text file over the established FTP connection.

The helper code for both tasks is written in JAVA, and the explanations are based on the provided Java code as well to help you throughout the assignment. However, this does not stop you from using Python. If you decide to use Python for implementing this same assignment, you will have to write all of the helper code yourself and fill in the missing code as well. In any case, follow a similar code organization and make sure you satisfy same requirements (e.g., your web server has to be multithreaded and has to handle requests of different types – images, text, html, embedded objects, etc).

You should **first** complete Task 1 by following the instructions under the "**Task 1**" section. Once you test your web server implementation from Task 1, follow the instructions in the "**Task 2**" section to implement the FTP client and extend your web server implementation.

Getting Started Hints For Java Implementation

If you decide to use Java, the easiest way to start is to create a project folder somewhere on your File System (this will be your root folder), then create another sub-directory under the root folder called webserver and place your implementation/class files (i.e., WebServer.java, FTPClient.java) under that directory.



Testing Your Code

Test your WebServer implementation for different types of requests using a ***variety of Web browsers (Chrome, Edge, Firefox, etc.).*** Your implementation may work for one browser but not the other, so make sure all browsers display the expected page.
Examples:

- a pure image request

- a pure html request
- a request for an html with embedded images
- a request for a non existing file
- a request for a missing.txt file (the missing file will be obtained from your FTP server and its content will be rendered in the web browser)

You may be asked to demonstrate the above scenarios at the bare minimum if a demonstration is required. So be prepared with at least the above scenarios in hand

HINT: your html files, images, etc. should all be placed under the root directory of your project, otherwise, your server will not find the file requested from the browser.

Submission

Submit **only your source code** on Canvas.

Submit the following two files:

- **WebServer.java** (The completed version of WebServer.java after all modifications made in task2)
Important Note: please **do NOT submit two versions of WebServer.java** for each task. You must submit ONLY one version for both tasks.

- **FtpClient.java** (the completed version of FtpClient.java)

Place both files in a **zipped** archive (use your **LAST NAME** as the archive name)

Grading (Total 25 points)

- Compiles and runs (2 points)
- Functionality / working scenarios (22 points)
- Displays entire http requests/responses in console as instructed (1 point)

Task1: Building a Multi-Threaded Web Server

We will develop a Web server in two steps. In the end, you will have built a multi-threaded Web server that is capable of processing multiple simultaneous service requests in parallel. You should be able to demonstrate that your Web server is capable of delivering your home page to a Web browser.

We are going to implement version 1.0 of HTTP, as defined in RFC 1945, where separate HTTP requests are sent for each component of the Web page. The server will be able to handle multiple simultaneous service requests in parallel. This means that the Web server is multi-threaded. In the main thread, the server listens to 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 that it receives. After this program is running properly, you will add the code required to generate an appropriate response.

As you are developing 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 that you give to your browser. For example, if your machine's name is `host.someschool.edu`, your server is listening to port 6789, and you want to retrieve the file `index.html`, then you would specify the following URL within the browser:

http://host.someschool.edu:6789/index.html

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

If you run the server program on the same physical machine as your web-browser, you can simply use the keyword “localhost” for the machine name:

http://localhost:6789/index.html

When the server encounters an error, it sends a response message with the appropriate HTML source so that the error information is displayed in the browser window.

Web Server in Java: Part A

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.

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`. The structure of the Web server is shown below (**IMPORTANT**: Remember to write your name at the top of the file):

```
/**
* Assignment 1
* Student Name
**/
import java.io.* ;
```

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

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

final class HttpRequest implements Runnable
{
        . . .
}
```

Normally, Web servers process service requests that they receive through well-known port number 80. But for this assignment, you will develop your own web server that runs on an arbitrary port number that is higher than 1024. Most of these ports are called user ports and can be used by application developers. **IMPORTANT**: Even though you can use any number between 1024 and 49,151 as the port number, please use port **6789** to make it easier to grade your submission. Also remember to use the same port number when making requests to your Web server from your browser via the URL field, as discussed later.

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

        . . .
}
```

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 of an infinite loop. This means 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.
        ?

        . . .
}
```

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.

```
// 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(request);

// Start the thread.
```

```
thread.start();
```

In order to have the `HttpRequest` object handle the incoming HTTP service 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.

After the new thread has been created and started, execution in the 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()`. For the remainder of the lab, it remains to develop the `HttpRequest` class.

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. 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()
        {
                . . .
        }

        private void processRequest() throws Exception
        {
                . . .
        }
}
```

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 can not 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 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
{
        // Get a reference to the socket's input and output streams.
        InputStream is = ?;
        DataOutputStream os = ?;

        // Set up input stream filters.
        ?
        BufferedReader br = ?;

        . . .
}
```

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. (See Section 2.2 of the textbook for a description of this and the following fields.)

```
// Get 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 display the header lines.
String headerLine = null;
while ((headerLine = br.readLine()).length() != 0) {
        System.out.println(headerLine);
}
```

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. (See the HTTP Request Message diagram in Section 2.2 of the textbook)

In the next step of this lab, 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. Add the following lines of code to close the streams and socket connection.

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

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 IP address of your running server. For example, if your machine name is `host.someschool.edu`, and you ran the server with port number 6789, then you would specify the following URL:

```
http://host.someschool.edu:6789/
```

The server should display the contents of the HTTP request message. Check that it matches the message format shown in the HTTP Request Message diagram in Section 2.2 of the textbook.

Web Server in Java: Part B

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 are going to ignore the information in the header lines, and use only the file name contained in the request line. In fact, we are going to 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.

```
// Extract the filename from the request line.
StringTokenizer tokens = new StringTokenizer(requestLine);
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.
fileName = "." + fileName;
```

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 = new FileInputStream(fileName);
} 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 are going to 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>";
}
```

IMPORTANT: Just like with the request message (i.e. GET message) made by the browser, your WebServer MUST also **print** the ENTIRE formed ***response*** message on the **console**/screen to help graders verify that you have the correct response message prior to sending it back to the client's browser. (no need to show entity body except for 'Not Found' similar to what is shown in the code above). So your screen should display something like this for each request/response pair every time the client/browser requests a certain file from the server:

Request:

…
Response:

…

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(statusLine);

// Send the content type line.
```

```
os.writeBytes(?);

// Send a blank line to indicate the end of the header lines.
os.writeBytes(CRLF);
```

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 that 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(buffer, 0, bytes);
   }
}
```

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";
        }
        if(?) {
                ?;
        }
        if(?) {
                ?;
        }
        return "application/octet-stream";
}
```

There is a lot missing from this method. For instance, nothing is returned for GIF or JPEG files. You may want to add the missing file types yourself, so that the components of your home page are sent with the content type correctly specified in the content type header line. For GIFs the MIME type is `image/gif` and for JPEGs it is `image/jpeg`.

This completes the code for the second phase of development of your Web server. Try running the server from the directory where your home page is located, and try viewing your home page files with a browser. Remember to include a port specifier in the URL of your home page, so that your browser doesn't try to connect to the default port 80. When you connect to the running web server with the browser, examine the GET message requests that the web server receives from the browser.

Task2: Implementing the FTP Client

A) First you need to install an FTP server on your local machine. FTP is a **very insecure** protocol to use for transferring files. Do not attempt to use the same setup in this programming assignment for managing your own files. Only use this setup for completing this assignment.

Since FTP is insecure, we will use Docker for this assignment to better protect your physical machine. Docker allows you to create an isolated environment "called a container". You can operate in this environment in isolation from your physical machine and operation system. If you do not know what Docker is, I encourage you to read more about it here: https://docs.docker.com/get-started/overview/.Follow the steps below to install an FTP server on your local machine:

1) Download Docker Desktop from here: https://www.docker.com/products/docker-desktop

2) Click on the downloaded file and follow the on-screen instructions to install Docker.

3) Once Docker is installed "*if using MAC OS, you may need to allow access to filesystem even after completing installation before you can proceed*", you need to install an FTP Server docker container. To do so, open a terminal or command line window and run the following command (**Remember** to replace **<USER_NAME>** and **<PASSWORD>** with your **own chosen credentials**. *Do NOT choose a password that you regularly use for logging into your sensitive accounts online (e.g., bank accounts, social media, etc.). Choose a fake password just for this programming assignment.*

> docker run --rm -it -p 20:20 -p 21:21 -p 4559-4564:4559-4564 -e FTP_USER=**<USER_NAME>** -e FTP_PASSWORD=**<PASSWORD>** docker.io/panubo/vsftpd:latest

Notice the different **-p** options above. These are used for mapping ports on your local machine to ports inside the container. Please read here (https://docs.docker.com/config/containers/container-networking/) for more details.

4) Open a **new terminal** and list all docker containers in order to find the **id** of the container you just installed. We will use this in the next step.

> docker container ls

5) Now we will log in to the container we just created and change the ownership of the ftp directory in order to be able to write files onto it. Replace **<CONTAINER_ID>** below with the CONTAINER ID from the previous step.

> docker exec -it **<CONTAINER_ID>** /bin/bash

6) If step 5 is successful, you should get a terminal inside the container (something like: root@194ea06a23fd:/#). Now change the ownership of the ftp directory as follows.

root@194ea06a23fd:/# **chown ftp:ftp /srv**

B) You are done setting up your FTP Server on your local machine. Now test your server by downloading FileZilla Client (https://filezilla-project.org/download.php?platform=win64) and use it to connect to the server. Remember to enter the user name and password you chose in **step 3**.

On your local machine, create a simple text file 'ftp_test.txt' with some content and then upload it to the FTP server using the FileZilla Client. This file can be used for testing your FTPClient code later.

If you do not know how FileZilla Client works to connect to an FTP server and upload files, you may follow this tutorial (https://wiki.filezilla-project.org/FileZilla_Client_Tutorial_(en)).

C) Once your FTP server is installed successfully. Move on to implementing the FTP client:
To make things easier for you, if you writing your code in Java, I am providing an incomplete **FtpClient.java** class under the assignment module on canvas. All you need to do is download the file, add it to your project, and fill in the missing snippets of code.

For the full list of native FTP commands and reply codes please refer to RFC 959 (http://www.ietf.org/rfc/rfc959.txt).

NOTE 1: To enable debugging messages set the DEBUG flag to true

NOTE 2: There are two data transfer modes in FTP, active and passive, please consult RFC 959 for further details on how each mode works. We will use the passive mode in this assignment. For the passive mode to work, you may need to disable IPv6 feature in your OS if it is enabled. Use this link (http://support.microsoft.com/kb/929852) to do so if you are running the FTP client on Windows machine.

NOTE 3: Make sure you turn off your firewall if your FtpClient is refusing to connect to FTP server

To test your FtpClient separately, you can create a test class with a main function. Try to instantiate an FtpClient, connect using your FTP user name and password, and download a file that you already uploaded to your FTP home directory:

```
package webserver;
public class FtpTest {
   public static void main(String args[]) {
     ...
   }
}
```

D) Extend your web server to act as an FTP Client when requesting text files only (.txt). Mainly, when you request a text file (.txt) from within your web browser, the web server will not have a copy of this file. It will instantiate an FtpClient, retrieve the text file from your local FTP server and then send it back to your web browser as an HTTP response.

More specifically, you will need to modify the following snippets of code:

```
// Construct the response message.
```

```
String statusLine = null;
String contentTypeLine = null;
String entityBody = null;
if (fileExists) {
        statusLine = ?;
        contentTypeLine = "Content-type: " +
                contentType( fileName ) + CRLF;
} else {
        // if the file requested is any type other than a text (.txt) file, report
        // error to the web client
        if (!contentType(fileName).equalsIgnoreCase(?)) {
             statusLine = ?;
             contentTypeLine = ?;
         entityBody = "<HTML>" +
                      "<HEAD><TITLE>Not Found</TITLE></HEAD>" +
                      "<BODY>Not Found</BODY></HTML>";
        } else { // else retrieve the text (.txt) file from your local FTP server
                statusLine = ?;
                contentTypeLine = ?;

             // create an instance of ftp client
             ?

             // connect to the ftp server
             ?

             // retrieve the file from the ftp server, remember you need to
             // first upload this file to the ftp server under your user
             // ftp directory
             ?

             // disconnect from ftp server
             ?

             // assign input stream to read the recently ftp-downloaded file
             fis = new FileInputStream(fileName);
        }
}

// Send the entity body.
if (fileExists) {
        sendBytes(fis, os);
        fis.close();
} else {
     if (!contentType(fileName).equalsIgnoreCase(?)) {
           os.writeBytes(?);
        } else {
           sendBytes(fis, os);
        }
}
```