CS 1652 – Data Communication and Computer Networks – Project#1[1]

Due: Friday June. 14th @ 11:59pm

All supplied source files, after filling in the missing parts, should be submitted properly on Gradescope. The places where you need to fill in code are marked with /* Fill in */.

Late submission: Sunday June 16th @11:59pm with 10% penalty per late day

OVERVIEW

Purpose: To use Java Sockets, File I/O, Input and Output Streams, and Threads, and to practice working with the HTTP protocol.

Goal 1: To implement a simple multi-threaded Web Server.

Goal 2: To implement a Web Proxy/Cache.

More specific details follow below.

IMPORTANT NOTE

For better security, run the Web Server and the Web Proxy that you will develop in this project in an environment that is isolated from incoming connections from the Internet, such as in a Virtual Machine.

DETAILS

DETAILS 1: MULTI-THREADED WEB SERVER IN JAVA

In this part you 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.

You 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.

[1] Adapted from Computer Networking: a Top-Down Approach, 7th Edition.

To simplify this programming task, you are encouraged to 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 virtual machine's IP address is a.b.c.d, 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://a.b.c.d: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 the browser runs on the same machine as the Web Server, you can use the URL: 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.

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 HttpRequestRunnable. The structure of the Web server is shown below:

```
import java.io.*;
import java.net.*;
import java.util.*;
public final class WebServer
{
        public static void main(String argv[]) throws Exception {
                . . .
        }
}
final class HttpRequestRunnable implements Runnable {
        . . .
}
```

Normally, Web servers process service requests that they receive through the 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.

```
// Set the port number.
private final static 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 HttpRequestRunnable 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.
HttpRequestRunnable request = new HttpRequestRunnable(?);

// Create a new thread to process the request.
Thread thread = new Thread(request);

// Start the thread.
thread.start();
```

In order to have the HttpRequestRunnable 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 HttpRequestRunnable 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 this part, it remains to develop the HttpRequestRunnable class.

We declare two variables for the HttpRequestRunnable 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 HttpRequestRunnable class is shown below:

```
final class HttpRequestRunnable implements Runnable
{
        final static String CRLF = "\r\n";
        private Socket clientSocket;
        HttpRequestRunnable(Socket socket) {
                clientSocket = socket;
        }
        @Override
        public void run() {
                . . .
        }
        private void processRequest() throws Exception {
                . . .
        }
}
```

In order to pass an instance of the HttpRequestRunnable class to the Thread's constructor, HttpRequestRunnable 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.

```
@Override
```

```
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, 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();
clientSocket.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. You may also use the following URL:

http://localhost: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.

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.

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.

```
// 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>";
}
```

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.

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.

```
private static String contentType(String fileName)
{
        if(fileName.endsWith(".htm") || fileName.endsWith(".html")) {
                return "text/html";
        }
        if(?) {
                return ?;
        }
        if(?) {
                return ?;
        }
        return "application/octet-stream";
}
```

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.

Details 2: Multi-Threaded Web Proxy/Cache

In this part you will develop a small web proxy server which also caches web pages. This is a very simple proxy server which only understands simple GET and POST requests, but is able to handle all kinds of objects, not just HTML pages, but also images.

Code (Skeleton provided for you on CourseWeb)

The code is divided into three classes as follows:

- ProxyCache holds the start-up code for the proxy and code for handling the requests.
- HttpRequest contains the routines for parsing and processing the incoming requests from clients.
- HttpResponse takes care of reading the replies from servers and processing them.

Your work will be to complete the proxy so that it is able to receive requests, forward them, read replies, and return those to the clients. You will need to complete the classes ProxyCache, HttpRequest,

and HttpResponse. The places where you need to fill in code are marked with **/* Fill in */**. Each place may require one or more lines of code.

NOTE: As explained below, the proxy uses DataInputStreams for processing the replies from servers. This is because the replies are a mixture of textual and binary data and the only input streams in Java which allow treating both at the same time are DataInputStreams. To get the code to compile, you may either use the -deprecation argument for the compiler as follows: javac -deprecation *.java or add the compiler annotation @SuppressWarnings("deprecation") before the constructor of the HttpResponse and HttpRequest classes.

If you do not use either option, the compiler may refuse to compile your code!

Running the Proxy

Running the proxy is as follows:

java ProxyCache *port*

where *port* is the port number on which you want the proxy to listen for incoming connections from clients.

Configuring Your Browser

You will also need to configure your web browser to use your proxy. This depends on your browser. In Internet Explorer, you can set the proxy in "Internet Options" in the Connections tab under LAN Settings. In Netscape (and derived browsers, such as Mozilla and Firefox), you can set the proxy in Edit->Preferences and then select Advanced and Proxies. In Google Chrome, the option is under Settings -> System -> Open Proxy Settings -> Web Proxy (HTTP).

In all cases you need to give the address of the proxy and the port number which you gave when you started the proxy. You can run the proxy and browser on the same computer without any problems, in which case the proxy address can just be localhost or 127.0.0.1.

Proxy Functionality

The proxy works as follows.

1. The proxy listens for requests from clients
2. When there is a request, the proxy spawns a new thread for handling the request and creates an HttpRequest object which contains the request.
3. The new thread sends the request to the server and reads the server's reply into an HttpResponse object.
4. The thread sends the response back to the requesting client.

Your task is to complete the code which handles the above process. Most of the error handling in the proxy is very simple and it does not inform the client about errors. When there are errors, the proxy will simply stop processing the request and the client will eventually get a timeout.

Some browsers also send their requests one at a time, without using parallel connections. Especially in pages with lot of inlined images, this may cause the page to load very slowly.

Caching

The basic functionality of caching goes as follows.

1. When the proxy gets a request, it checks if the requested object is cached, and if yes, then returns the object from the cache, without contacting the server.
2. If the object is not cached, the proxy retrieves the object from the server, returns it to the client, and caches a copy for future requests.

In practice, the proxy must verify that the cached responses are still valid and that they are the correct response to the client's request. You can read more about caching and how it is handled in HTTP in RFC 2068. For this project, it is sufficient to implement the above simple policy. You do not need to implement any replacement or validation policies.

Your implementation will need to be able to write responses to the disk (i.e., the cache) and fetch them from disk when you get a cache hit. For this you need to implement some internal dictionary data structure in the proxy to keep track of which objects are cached and where they are on disk. You can keep this data structure in main memory; there is no need to make it persist across shutdowns.

POST Requests

Add support for POST, by including the request body sent in the POST-request. You can test that by entering a search query inside the search box on the top-right corner of the webpage at http://www.pitt.edu and hitting ENTER or clicking on the search icon.

Programming Hints

Most of the code you need to write relates to processing HTTP requests and responses as well as handling Java sockets.

One point worth noting is the processing of replies from the server. In an HTTP response, the headers are sent as ASCII lines, separated by CRLF character sequences. The headers are followed by an empty line and the response body, which can be binary data in the case of images, for example.

Java separates the input streams according to whether they are text-based or binary, which presents a small problem in this case. Only DataInputStreams are able to handle both text and binary data simultaneously; all other streams are either pure text (e.g., BufferedReader), or pure binary (e.g., BufferedInputStream), and mixing them on the same socket does not generally work.

The DataInputStream has a small gotcha, because it is not able to guarantee that the data it reads can be correctly converted to the correct characters on every platform (DataInputStream.readLine() function). In the case of this lab, the conversion usually works, but the compiler will flag the DataInputStream.readLine() method as deprecated and will refuse to compile without the -deprecation flag.

It is highly recommended that you use the DataInputStream for reading the response.

EXTRA CREDIT

Better error handling (10 points). Currently the proxy does no error handling. This can be a problem especially when the client requests an object which is not available, since the "404 Not found" response usually has no response body and the proxy assumes there is a body and tries to read it. Modify your proxy to return appropriate error messages to the client instead of causing it to just time out.

Accessing your Web server using your Web Proxy (5 points). Try accessing http://localhost:6789/index.html (or whatever URL worked when testing your Web server) while configuring the browser to use your Web proxy. Make the necessary modifications to make both programs work together correctly.

SUBMISSION REQUIREMENTS

You must submit the following complete, working source files for full credit:

1. WebServer.java
2. ProxyCache.java
3. HttpRequest.java
4. HttpResponse.java

The idea from your submission is that your TA (and the autograder) can compile and run both of the main programs (WebServer.java and ProxyCache.java) **from the command line** WITHOUT ANY additional files or changes, so be sure to test it thoroughly before submitting it.

Note: If you use an IDE such as NetBeans, Eclipse, or IntelliJ, to develop your programs, make sure they will compile and run on the command line before submitting – this may require some modifications to your program (such as removing some package information).

GRADE BREAKDOWN

Item	Number of Points
Web Server	40
Basic Web Proxy	30
Caching	20
POST Request handling	10
EXTRA Credits	15
MAX Points	**100**