Lab#10
CSCI 201

Title
Web Server

Lecture Topics Emphasized
Thread Methods
Networking Theory
Networking Programming

Introduction
A web server is a very simple application that receives requests from a client (typically a web browser), and sends back out the contents of a file, if it exists. A web server differs from an application server in that it only sends the content of a file back but does not do any server-side processing. This just requires accepting a connection from a client, parsing the messages that is passed to it, and responding with an appropriate file.

Description
We have been using Tomcat throughout the semester. Essentially, Tomcat handles all incoming HTTP requests, and finds and returns the requested files if they exist. However, in this lab, you will be writing a native server, which does Tomcat's job (exciting!).

Assuming you run your web server on port 6789, to request the page `test.html`, you will load the page http://localhost:6789/test.html in a browser. Your web server will then try to find the file `test.html`. If it finds it, it will send the content of that file back to the browser. If that file contains HTML, CSS, and JavaScript, the browser will interpret the code and display it just like a regular web site. If the file cannot be found, send back an HTTP 404 message in the header of the HTTP response. The browser will then display the default 404 message.

You will need to allow as many clients to connect to your web server as possible (requiring multi-threading). Each connection should be handled by a separate thread. Refer back to the server example (*ChatRoom.java* and *ServerThread.java*) we did in the multithreaded lecture to give you a better idea of what we are doing in this lab. The code you will be writing today will resemble the two Java classes that we wrote in class.

Please download the skeleton code under the lab assignment, and add it to an Eclipse project. Some of the steps in this lab instruction have already been completed for you in the skeleton project. There is no need to redo those steps or to create duplicate files.

In your backend *src* folder, add the *Server* class. In the *Server* class constructor, you will need to instantiate a new *ServerSocket*, and you may bind the socket to any unused port that you would like (most likely one above 1023). Since the server will need to be constantly and repeatedly accepting new connections, do the following steps in an infinite while loop. The *ServerSocket* will be accepting client connections using the *accept()* method. Upon receiving a new connection, the server will need to create a new *ServerThread* (which we will be writing in a moment) to handle that request. Remember to write create a main method in your *Server* class, where you will create a new instance of the server.

Now, it's time for us to write the *ServerThread* to handle the request. Modify the *ServerThread* that extends from *Thread*. The thread will need to keep track of the client socket that connected to the server, so make sure you pass the socket into the thread's constructor, and create a *Socket* variable to keep track of it. You will also need to create instance variables for *PrintStream* and *BufferedReader,* which we will be using later to determine the content of the request and write response back. In the *ServerThread* constructor, instantiate the two variables. You can get a new instance of *PrintStream* from the socket OutputStream by calling *new PrintStream(new BufferedOutputStream(socket.getOutputStream()))*. Similarly, instantiate the *BufferedReader* from the socket InputStream by using *new BufferedReader(new InputStreamReader(socket.getInputStream())).* Make sure you start your thread in the constructor.

In your *ServerThread* run method, you will need to parse the request and send the appropriate response back to the clients. Clients will be connecting to your web server using a web browser. That means that you need to communicate using HTTP. You can look up online what the header of an HTTP request looks like. Use the *BufferedReader* to read the request header. Once you parse the incoming request, you will try to find the file that was requested. Before writing the response, create a response header based on the file type that was requested. Print the response header to the *PrintStream* that we created earlier in the constructor. If the file requested is there, read in the contents of the file through an *InputStream*, then pass the streams of the file from the *InputStream* to the *PrintStream* byte by byte. If the file requested is invalid, respond with a 404 page. Make sure you close the *PrintStream* before exiting the run method.

You can put a few test HTML files into a directory on your computer where your web server will check. Once your web server is working, try adding images into the directory, and modify your HTML files to include `img` tags. The browser will automatically request the image files, and you will just need to change the content type of the HTTP header to represent an image being returned, followed by the contents of the image file.

Grading Criteria

Labs are graded based on your understanding of the course material. To receive full credit, you will need to 1) complete the lab following the instructions above **AND** 2) show your understanding of the lab material by answering questions upon check-off.

If there is a discrepancy between your understanding of the material and your implementation (i.e. if your code is someone else's work), you will receive a grade of **0** for the lab. Please note, it is the professor's discretion to report the incident to SJACS.

Instructors, to ensure consistency across all lab sections, please strictly stick to the following criteria:

1. *Server* class (25%)

a) 25% - the class is complete and working, following the instructions above
b) 20% - the class is complete (following the instructions above), but has bugs
c) 15% - the student is on the right track (following the instructions above, but the implementation is incomplete (has more than 50% done)
d) 0% - the student implements less than 50%

2. *ServerThread* Constructor (25%)

a) 25% - the constructor is complete and working, following the instructions above
b) 20% - the constructor is complete (following the instructions above), but has bugs
c) 15% - the student is on the right track (following the instructions above, but the implementation is incomplete (has more than 50% done)
d) 0% - the student implements less than 50%

3. *ServerThread* Run Method (25%)

a) 25% - the run method properly handles HTTP requests (any file type is okay, but hard-coded response does not count)
b) 20% – the run method is ***complete*** and able to correctly identify the requested files, but has bugs in writing responses
c) 15% - the run method properly parses HTTP requests, and correctly identifies the requested files
c) 0% – the run method is unable to identify the requested files (***no partial credits***)

4. Check-off Questions: Please randomly select three questions from the following (25%).

   a) Where do we accept new connections in this lab?
   b) How do we make sure that the server is constantly accepting new connections?
   c) How does the server handle multiple requests at the same time?
   d) Why do we parse requests and write responses in threads?
   e) How do we identify the requested files?

f) How do we write the responses back to the clients?
g) How does HTTP come into play?
h) What do we need to do before writing the content of the request file to the *Printstream*?

Submission

Please export your Eclipse project following the instructions in the first lab (the output will be a .zip file) and answer the check-off questions in a separate pdf (your lab CPs will select three questions for you to answer).