CSDS 325/425
Project 1: HTTP 1.1 server
325 students: 100 pts + 10 bonus points
425 students: 120 points
Due: November 2 before 11:59pm.

In this project, you will develop a simplified HTTP 1.1 server using Java and test its operation using a standard browser. Since everyone will be sharing the same machine, you will need to have a unique port number (since, as you know, no two processes on the same host can listen to the same port). So, please use port 50000+x, where x is your order number in the combined class list (uploaded to canvas under "files").

Your server needs to be able to:

(1) Accept a connection request from an external browser and read the request
(2) Extract the requested URL and header fields that are needed for your server; you can ignore the header fields your simplified server does not need.
(3) Produce the HTTP response, send it back to the browser, and close the connection (i.e., your server will be using non-persistent HTTP). Note: don't implement the full HTTP protocol. Just read the headers you need from the request (ignoring the rest), and return the minimal set of headers sufficient for the browsers to render your content.
(4) Any URLs hosted by your server must have the format http://eecslab-10.case.edu:<your_port_#>/<your_Case_ID>/*. E.g., my server would have URLs http://eecslab-10.case.edu:50061/mxr136/*. Any request for a non-compliant URL must return "404 Not Found" response. I will use "mxr136" and my port number throughout the project description below, but you should replace it with your Case ID and port number. Note: you need to log in to Case VPN to login to the eecslab-10 machine.
(5) Your server must support three URLs listed below. Any requests for any other URLs must produce "404 Not Found" response.
    a. A special URL, http://eecslab-10.case.edu: 50061/mxr136/visits.html that returns an HTML page specifying the number of visits a given browser has made to any valid URLs on your web server. Notes: (i) the requests producing errors (such as "404 Not Found") should not contribute to this count; (ii) Make it a simple HTML page, nothing fancy, e.g., just the text "Your browser visited various URLs on this site X times" would do. This response will be dynamically generated upon receiving the above URL. Use HTTP cookies to implement this functionality. Tip: since everyone in the class uses the same host, you must be careful not to count browser visits to someone else's website. You will need to use cookie attributes to limit its scope to just your webserver, as well as to ensure that the cookie is persistent enough for your purpose. See, e.g., https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie for a good description of cookie attributes.
    b. A URL http://eecslab-10.case.edu:50061/mxr136/test1.html that will return an HTML object served from a statically stored file, which you will create using your

favorite regular text editor like emacs, vi, etc. (Note: don't use HTML authoring tools which produce very complicated HTML code that is all but impossible to read. We will be reading your HTML code. The pages I am asking you to create are at the trivial "Hello, World" level – you don't need to learn full HTML. There are lot of tutorials that would allow you to do this quickly. E.g., go through examples in https://www.learn-html.org/en/Hello%2C_World%21 and click on "links" and "images" from there.) This object needs to include: (i) a header "This is test page 1" (don't confuse this use of the word "header" with protocol headers; in this context, the header is like a document title or section header in a document, i.e., part of the page content) (ii) the text "Hi my name is <Your name>. Below is an external image I chose to embed in this page and a hyperlink to an external website.", (iii) an embedded image from an external website, and (iv) the word "hyperlink" in the above sentence being an actual clickable hyperlink to an external page. Make sure that when you access this HTML page from a standard browser (e.g., Chrome), the browser displays your embedded image and the hyperlink, and you can click on the hyperlink to go to the indicated external page.

c. A URL http://eecslab-10.case.edu:50061/mxr136/test2.html, another static HTML page that you must write manually using a text editor that includes (i) a header "This is test page 2", (ii) the text "Hi, my name is <Your name>. Below is the iframe that embeds my test page 1, and (3) an iframe (an "inlined frame") with borders showing your test page 1. An iframe is an HTML tag that allows one page to embed another page. The attributes of the iframe tag specify the URL of the embedded page, the dimensions of the area showing the embedded page, and sopme optional style parameters such as borders. See https://www.w3schools.com/tags/tag_iframe.ASP for details and simple examples, which will suffice for this step.

(6) Your server should be stateless – it should not store any information about HTTP requests to count browser accesses. Note: you don't have to handle potential race conditions that may occur when a user simultaneously requests URLs from your server from multiple tabs or windows. In other words, assume that a browser only accesses your website from a single tab or window.

(7) Don't hard-code the port number your server will be using and the directory from where the server will be reading static files. Instead, have a simple configuration file that the server will read when it's started and which specifies this information.

(8) Your server must be multi-threaded, e.g., use a separate thread to process each request and therefore be able to process multiple requests concurrently. For 325 students, the server does not need to support persistent connection: each thread closes its connection socket after processing the request. Tip: since HTTP 1.1 assumes persistent connections by default, your response must specify explicitly that you are closing the connection, using the Connection header. See https://tools.ietf.org/html/rfc7230 for details.

425 part:

Your server must include the option to support persistent HTTP. Note: without persistent connections, the server can simply indicate the end of the response by closing the connection. With persistent connections, it needs to provide this indication through other means, which in HTTP 1.1 is basically done using a Content-length header field (and it may be easier to just insert Content-length always, whether or not persistent connections are used). See https://tools.ietf.org/html/rfc7230 for details. Please add a line in the configuration file that specifies whether your server will support persistent connections, and for how long it will keep the idle connection open. We will test your server under both regimes, with and without persistent connections.

Deliverables:

(1) Well-commented source code for the implementation of your server. Please document your code with embedded comments within the code, not automatically generated specification of various methods (like what IDEs like Dr. Java would produce). Deployed server on eecslab-10.case.edu machine. Your directory on this machine must include the source code for your server, the configuration file, the two static HTML files, and a README file in your home directory that specifies (a) the location of the above information (the source code, the config file, and the HTML files) within your home directory, (b) the command to start your server from the terminal window, and (c) which browser (name + version, e.g., Chrome Version 85.0.4183.121) you used to test your website.
(2) A zip file comprising the above information submitted to canvas

Grading rubrics (325 students: 100 pts + 10 bonus pts for 425 part; 425 students: 120 pts):

- Programming style (code is logical and well commented, i.e., understandable): 10
- Connection management (handling welcome and connection sockets, closing of sockets): 15
- Multithreading: 15
- Cookie management (including correct set-cookie headers in responses that support access-counting without keeping state, reading and interpreting cookie headers in requests): 15
- Request and request headers processing: 10
- Dynamic HTML response generation/transmission: 10
- Static HTML responses generation/transmission: 10
- HTML code: 10
- Configuration file processing: 5
- (425) Implementation of persistent connections (including extension to the configuration file and timeout management): 20 pts (10 bonus points for 325 students)

Tips:

1. You can easily debug your HTML code locally by just opening the HTML file with a browser (e.g., on my Mac, I can just double-click on the file name in the Finder window and it opens with a default browser). However, for the browser to find embedded URLs, these URLs need to be in a so-called relative form, i.e., specified starting from the directory in which the container page resides. E.g., if both test1.html and test2.html files are in the same directory, you can specify the iframe URL as "./test1.html".
2. You can also debug your web server on a local machine by accessing its URLs from a browser that also runs on the local machine. Just replace the hostname "eecslab-10.case.edu" in all URLs with "localhost". E.g., the test1 page would have URL http://localhost:50061/mxr136/test1.html.
3. (Please talk to the TAs if you have trouble understanding this tip): When debugging networked applications, it's sometimes useful to see what packets are actually sent and/or received by a given end-point. On your local machine, you can always run wireshark to record and view the packets in a nicely formatted way (or use tcpdump, the tool wireshark is built on, directly, for doing this from the terminal window rather than from a nice GUI). On a remote machine (like eecslab-10), running an application and displaying its GUI on your local machine is more difficult. Besides, you don't have root access to this machine needed to run wireshark or traceroute. This provides another argument for debugging your web server on your local machine, where you can run any tool you want. Just make sure you tell wireshark to sniff traffic on the so-called "loopback" interface (which is a "pretend" interface that handles traffic that actually does not leave the machine but just flows between the processes on this machine in your case the browser and the web server processes).

Important:

To avoid running into academic integrity issues, don't search the Internet for examples of Java webserver code. It's fine to look for examples of general issues but don't search for specific examples related to HTTP server implementation. E.g., it's OK to search how to read a line from a TCP connection socket but not OK to search for code that reads and parses an HTTP request. It's OK to search how to run a general function as a separate thread but not OK to search for code that issues those threads to process an HTTP request. Further, please strictly follow the academic integrity instructions on "Dos and Don'ts" from the syllabus, which I reproduce below for your convenience:

For projects, I basically adhere to the policy articulated by Prof. Connamacher, which I reproduce below with his permission (slightly edited). Here is a short list of things you may and may not do:

- DO NOT send an electronic copy of any part of your code to another student.
- DO NOT look at another person's code for the purpose of writing your own.
- DO NOT tell another student any code needed for a project.
- DO NOT search the internet to find an implementation of your project.

- DO NOT post code you write for this class onto an internet site such as StackExchange or github. If you find on the Internet a site that supports private (password-protected) repositories, it's OK to post your code to your private repository and specify both the URL and password in your resume. This way, your prospective employers will be able to see your work but other students won't.
- DO sit down with a student and go over the student's code together in order to help the student find a bug. However, when you find a bug, explain the issue but not a solution. For example, if you found an infinite loop, you can say "This loop will never reach the termination condition" but you can't say "Put 'i = i+1;" at the end of your loop." In other words, you can say what's wrong with the student's code but not how to make it right.
- DO describe in English (not code!) the general steps needed to solve a programming task.
- DO show students how to do certain coding tasks as long as the task can be viewed as general rather than specifically applicable to the project at hand. For instance, you can suggest a Java or Python statement for opening a network socket and setting various socket options, but you can't show or suggest a piece of code to construct a specific message to be sent from one process to another.
- DO use coding examples from the lectures and textbook as a guide in your programming.