COEN317: Distributed Systems
Winter 2024
Programming Assignment 1
Due: Tuesday, 2/6/2024 @11:59pm

Project Description

The goal of this programming assignment is to build a functional web server. This assignment will teach you the basics of distributed programming, client/server structures, and issues in building high performance servers. While the course lectures will focus on the concepts that enable network communication, it is also important to understand the structure of systems that make use of the global Internet.
This project should be done individually.

Part 1: Build the Server

At a high level, a web server listens for connections on a socket (bound to a specific *port* on a host machine). Clients connect to this socket and use a simple text-based protocol to retrieve files from the server. For example, you might try the following command from a UNIX machine:

```
$ telnet www.scu.edu 80
GET /index.shtml HTTP/1.0
```

(Type **two** carriage returns after the "GET" command). This will return to you (on the command line) the HTML representing the "front page" of the Santa Clara web page.

One of the key things to keep in mind in building your web server is that the server is translating relative filenames (such as `index.html`) to absolute filenames in a local filesystem. For example, you might decide to keep all the files for your server in `~username/coen317/server/files/`, which we call the **document root**. When your server gets a request for index.html (which is the default web page if no file is specified), it will prepend the document root to the specified file and determine if the file exists, and if the proper permissions are set on the file (typically the file has to be world readable). If the file does not exist, a file not found error (HTTP error code 404) is returned. If a file is present but the proper permissions are not set, a permission denied error is returned. Otherwise, an HTTP OK message is returned along with the contents of a file.

You should note that since `index.html` is the default file, web servers typically translate `'GET /'` to `'GET /index.html'`. That way, index.html is assumed to be the filename if no explicit filename is present. This is why the two URLs http://www.scu.edu and http://www.scu.edu/index.html return equivalent results. Also note that the default index.html page may actually redirect to a different (i.e.,

the real) home page. For example, in the case of www.scu.edu, requesting index.html returns a very short page that simply instructs the browser to request index.shtml instead. Normally your browser performs this redirection for you automatically, but a simple tool like telnet will simply show the actual contents of `index.html` if you request it.

HTTP/1.0 protocol

When you type a URL into a web browser, the server retrieves the contents of the requested file. If the file is of type *`text/html`* (i.e., it is a regular web page written in HTML) and the HTTP/1.0 protocol is being used, the browser will parse the HTML for embedded links (such as images) and then make separate connections to the web server to retrieve the embedded files. If a web page contains 4 images, a total of five separate connections will be made to the web server to retrieve the html and the four image files.

HTTP/1.1 protocol

Using HTTP/1.0, a separate connection is used for each requested file. This implies that the TCP connections being used never get out of the slow start phase. HTTP/1.1 attempts to address this limitation. When using HTTP/1.1, the server keeps connections to clients open, allowing for "persistent" connections and pipelining of client requests. That is, after the results of a single request are returned (e.g., index.html), the server should by default leave the connection open for some period of time, allowing the client to reuse that connection to make subsequent requests. One key issue here is determining how long to keep the connection open. This timeout needs to be configured in the server and ideally should be dynamic based on the number of other active connections the server is currently supporting. Thus if the server is idle, it can afford to leave the connection open for a relatively long period of time. If the server is busy servicing several clients at once, it may not be able to afford to have an idle connection sitting around (consuming kernel/thread resources) for very long. You should develop a simple heuristic to determine this timeout in your server.

For this assignment, you will need to support enough of both the HTTP/1.0 and HTTP/1.1 protocol to allow an existing web browser (e.g., Firefox) to connect to your web server and retrieve the contents of the Santa Clara University home page from your server. Of course, this will require that you copy the appropriate files to your server's document directory. Note that you DO NOT have to support script parsing (e.g., PHP or JavaScript), and you do not have to support HTTP POST requests. You should support images, and you should return appropriate HTTP error messages as needed.

At a high level, your web server will be structured something like the following:

Forever loop:
Listen for connections
    Accept new connection from incoming client
    Parse HTTP request

Ensure well-formed request (return error otherwise)
Determine if target file exists and if permissions are set properly (return error otherwise)
Transmit contents of file to connect (by performing reads on the file and writes on the socket)
Close the connection

You will have two main choices in how you structure your web server in the context of the above simple structure:

1. A multi-threaded approach will spawn a new thread for each incoming connection. That is, once the server accepts a connection, it will spawn a thread to parse the request, transmit the file, etc. If you decide to use a multi-threaded approach, you should use the pthreads thread library (i.e., `pthread_create`).

2. An event-driven architecture will keep a list of active connections and loop over them, performing a little bit of work on behalf of each connection. For example, there might be a loop that first checks to see if any new connections are pending to the server (performing appropriate bookkeeping if so), and then it will loop over all existing client connections and send a "block" of file data to each (e.g., 4096 bytes, or 8192 bytes, matching the granularity of disk block size). This event-driven architecture has the primary advantage of avoiding any synchronization issues associated with a multi-threaded model (though synchronization effects should be limited in your simple web server) and avoids the performance overhead of context switching among a number of threads. To implement this approach, you may need to use non-blocking sockets.

You may choose from C/C++, Java or Python to build your web server but you must do it in Linux (although the code should run on any Unix system). In C/C++, you will want to become familiar with the interactions of the following system calls to build your system: *socket(), select(), listen(), accept(), connect()* . Outlined below are a number of resources below with additional information on these system calls.

Part 2: Submission

Make the server document directory (the directory which the webserver uses to serve files) a command line option. The command line option must be specified as **-document_root**.
Make the port that the server listens on a command line option. The option must be specified as **-port**. Thus, I should be able to run your server as

```
$ ./server -document_root "/home/moazzeni/webserver_files" -port 8888
```

Note that you should use ports between 8000 and 9999 for testing purposes. While writing your server, you can test it from the same machine the server is running by using telnet.

To submit your assignment, create a tarball, e.g.:

```
tar czvf PA1-Firstname_Lastname.tar.gz your-project-files
```

which will create proj1-Firstname_Lastname.tar.gz from the files in your-project-files), and then upload proj1.tar.gz to Camino. You can use scp to copy from the machine you are writing your code on back to your local machine (or just use your version control system, e.g., Git, Subversion, etc).

Please include the following files in your tarball.

1. A Readme.txt file that should contain
   - Your name
   - Assignment name
   - High-level description of the assignment and what your program(s) does
   - A list of submitted files
   - Instructions for running your program
   - Include demonstration of your program's correct implementation. Show how you're making requests to the web server by include screenshots of a web browser accessing your web server.
2. All the files for your source code only. Please do not include any executable.
3. Your Makefile (if any or compile instructions).

Frequently Asked Questions

1. Do we need to send back HTTP status codes? If so, what status codes are required?
   Status codes are required. You should at least support the 200, 404, 403, and 400 status codes (and feel free to support others as well).
2. In addition to status codes, what other headers are required?
   At a minimum, you should support the Content-Type, Content-Length, and Date headers in your responses.
3. While files types do we have to support?
   Again, creativity is encouraged. However, you must at least support HTML, TXT, JPG, and GIF files.

Resources

Here is a list of available resources to help you get started with various aspects of this assignment. As always, Google and Linux man pages will also be useful.

- The classic C book (and a good reference book to have on your shelf): http://www.amazon.com/The-Programming-Language-2nd-Edition/dp/0131103628
- Socket Programming Tutorial: http://compnetworking.about.com/cs/socketprogramming/
- HTTP 1.0 and 1.1: http://www.jmarshall.com/easy/http/
- w3c HTTP page: http://www.w3.org/Protocols/
- Thread Programming Examples (C): http://www.cs.cf.ac.uk/Dave/C/node32.html
- C Programming link:

http://www.cs.cf.ac.uk/Dave/C/CE.html

- HTTP Wikipedia
  http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol