## Programming assignment 1: Building a web server

### Due: Monday, February 22nd, 2017

### 1. Description

#### 1.1 Part 1: Web Server

The goal of this assignment is to build a functional HTTP server. This assignment will teach you the basics of network 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. You can work in teams of two on this lab.

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.cs.csusm.edu 80

GET / HTTP/1.0\n

\n

(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 CSUSM   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 file system. For example, you may decide to keep all the files for your server in student/cs537/server/files/, which we call the  root. When your server gets a request for /index.html, it will prepend the  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 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 the file.

You should also note that web servers typically translate "GET /" to "GET /index.html". That is, index.html is assumed to be the filename if no explicit filename is present. The default filename can also be overridden and defined to be some other file in most web servers.

When you type a URL into a web browser, it will retrieve the contents of the file. If the file is of type text/html, it will parse the html for embedded links (such as images) and then make separate connections to the web  server to retrieve the embedded files. For example, 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. This discussion assumes the HTTP/1.0 protocol.

For this assignment, you will need to support enough of the HTTP protocol to allow an existing web browser (Firefox, Safari or Konqueror) to connect  to your web server and retrieve the contents of a sample page from your server.  (Of course, this will require that you copy the appropriate files to your server's document directory).

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 have three main design choices in how you structure your web server in the context of the above simple structure (*note that the event model  is required; the others give bonus points*):

* 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.
* A multi-process approach maintains a worker pool of active processes to hand requests off from the main server. This approach has the advantage of portability (relative to assuming the presence of a given threads-package across multiple hardware/software platforms). It does face increased context-switching overhead relative to a multi-threaded approach.
* 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 (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.

Finally, support the following commands:

* **GET:** Retrieves whatever information (in the form of an entity) is identified by the Request-URI
* **HEAD**: Asks for a response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.
* **POST**: Requests that the destination server accept the entity enclosed in the request.

Implement your assignment in either C or C++. You will want to become familiar with the interactions of the following system calls to build your system: socket(), select(), listen(), accept(), connect(). We outline a number of resources below with additional information on these system calls.  Several books are also available on this topic.

The format of the command line should be:

myhttpd [<http>]

[<port>]

If <http> is not passed, the server will run in HTTP/1.0 mode. If <port> is not passed, you will choose your own default port number. Make sure it is larger than 1024 and less than 65536.

#### 1.2 Part 2: Load Generator/Client

Now that you have a functional web server, the second part of the assignment involves evaluating the performance of the system that you have built.  Build a synthetic client program that connects to your server, retrieves a file in its entirety, and disconnects. The goal of this load generator is to evaluate the performance of your server. You will measure server performance in terms of throughput (requests/sec) and in terms of latency (average time to retrieve a file). Your synthetic load generator may be multi-threaded, with a different number of threads (this will be used to generate average throughput).

Consider the effects of the bandwidth delay product, round trip times, and file sizes on this tradeoff. For example, high round trip times exacerbate the negative effects of slow start (taking multiple rounds to send a file even if the bottleneck bandwidth would allow the entire file contents to be sent in a single round trip).

### 2. Report

A very important aspect of your assignment will be your report describing your system architecture, implementation, and high level design decisions. Make sure to clearly describe how you set up your experiments.  Finally, your write-up should include explicit instructions on how to install, compile, and execute your code..

### 3. Submission

You will write C++ (or C) code that compiles under the GCC (GNU Compiler Collection) environment. You have to make sure your code will *compile and run correctly on the* the CS department's Linux  machines. You should submit both your server and the client you used to test your server. Please remove all object files and submit only source codes with a make file. Add a readme.txt file describing how to compile and run your program from a terminal. Submit your assignment files, including your report, by the due date.

### 4. Grading Criteria and Demo

Students will demonstrate their projects after the due date of the assignment. Students should prepare a 5-10 minutes demo