COMP 112 Assignment 1: HTTP Server

Lead TAs: Eli Boninger and Julia Knight
Based on an assignment from Alva Couch
Tufts University

Due: Sept 26th, 2016 - 11:59pm

Introduction

In this assignment, you will write a simple web server that understands the HTTP protocol and responds to basic GET requests. Your server will respond to two URLs: one which returns a JSON dictionary describing IP addresses and the corresponding locations of recent connections to your page, and one which displays information about you. You will employ rate limiting to prevent users from overwhelming your server. These web pages will not correspond to files, but will be constructed by your program in response to each request. The server will handle one client at a time and should never quit.

Requirements

Write a program a1.c that does the following:

- Takes one command line argument:
  - A port number on which to listen for requests.
- Accepts TCP requests from web browsers/clients and responds with a single HTML page of content based on the URL:
  - The main page (/): An ordered list of the most recent 10 IP addresses that accessed it. Each IP address will be associated with the number of times the main page has been accessed from that specific address and the city where that IP originated from.
  - Another version of the main page (/): This is the page that will appear if a user is being rate limited. It will contain a simple warning message, and it will **not** be a HTTP 200 OK response.
  - An about page (about.html): Anything you'd like to say about yourself. It doesn't have to be long.
  - Return a proper HTTP/1.1 404 Not Found as the HTTP response if a URL other than "/" or "about.html" is requested.

The requirement for the main page is satisfied by keeping records in your server about the identities of the addresses that send you requests. You do not need to open files or databases; you can do this in memory. Restarting your server will cause it to clear its memory, and that's fine. The key to this assignment is to keep things simple and to utilize your understanding of the TCP sockets and HTTP protocol.

HTTP and HTML Responses

Every HTTP request is a stream of characters, separated by \n characters, which describe parts of the request. Many of these characters describe the browser that is asking for information.

Similarly, your response should be a stream of characters. The simplest HTTP response is:

```
HTTP/1.1 200 OK
Content-type: text/plain

hello world
```

The first line is the protocol description, which defines how your program intends to communicate in the subsequent lines. The second line is the content type, which says you will respond in plain text. The blank line is a delimiter. It is required between the header and the payload. Technically, there are two \n characters here. The hello world is the text that the browser should display.

To serve HTML, this pattern needs to be modified slightly:

```
HTTP/1.1 200 OK
Content-type: text/html

<html><body><h1>hello world</h1></body></html>
```

To serve JSON, a response might look like this:

```
HTTP/1.1 200 OK
Content-type: application/json

[ "hello world", 5, { "key1": "value1", "key2": "value2" } ]
```

The browser details should all be in the HTML and JSON responses your web server creates. The server doesn't need to output anything in the terminal, and we won't grade anything that shows up there. We will only be grading the HTML and JSON responses we receive when trying to visit a "page" on your web server.

Please note: we are not looking for beautiful web pages. Do not spend time trying to properly return CSS in your response.

Responding to Multiple URLs

While keeping your program self-contained, it should respond to two different URLs with different results. One will provide IP address statistics, while the other will provide a page about you. Completing these requests will require interpreting the HTTP GET portion of the request.

Sample Main Page Output

The follow is a sample main page sent by your server:

```
[{ "IP": "12.34.56.78", "Accesses": 6, "City": "Everett"}, {
"IP": "23.45.67.89", "Accesses": 3, "City": "Huntington"}, {
"IP": "7.65.4.12", "Accesses": 1, "City": "Columbus (East
Columbus)"}]
```

Formatted, it would look like:

```
[{
     "IP": "12.34.56.78",
     "Accesses": 6,
     "City": "Everett"
}, {
     "IP": "23.45.67.89",
     "Accesses": 3,
     "City": "Huntington"
}, {
     "IP": "7.65.4.12",
     "Accesses": 1,
     "City": "Columbus (East Columbus)"
}]
```

Note: **Your response will not be formatted in the above way - this is just meant to clearly show all the information you are returning.**

This indicates that the user with IP address 12.34.56.78 is located in Everett and has requested the main page six times, and so on. Your response should also be ordered such that the most recent user to access your page should be the first one on the list. In this case, the IP from Everett was the IP that most recently accessed your site.

Because your main page only lists the 10 most recent users to access it, it is possible that some users will be pushed out of that list if they have remained inactive for a long enough time. In the event that a user pushed off the list reconnects, you can restart their access count at zero.

IP Geolocating

You will need to use the freegeoip.net API to geolocate an IP (API documentation can be found at the website). If the API fails to detect the city that an IP originates from, you must check for this and send an appropriate error message in the JSON. For example, it would be appropriate to send this data for the IP address 127.0.0.1:

```
{    "IP": "127.0.0.1",
     "Accesses": 1,
     "City": "N/A" }
```

If the API call doesn't work for whatever reason (their server could be down), you should handle this and send a similarly appropriate error message.

You have to use socket programming and a GET request to do the IP geolocation.

Rate Limiting

You should keep track of the number of times each IP address has accessed your main page. You will also need to store some data about the times at which the most recent accesses occurred. You can use the sys/time.h library to retrieve UTC time from within a C program.

When a user returns to the main page over five times in under 20 seconds (assuming that users are uniquely identified by their IP address), instead of sending the regular IP list to the browser, you should send a warning message. The response code in this instance **should not be 200** but something more appropriate. The text of the warning is up to you.

If an IP has been rate limited previously, they should continue to get the error message as long as they keep requesting the page more than 5 times every 20 seconds. Once they fall below that rate, they should get the usual list of IPs again.

If an IP is currently rate limited, but their IP is pushed off the list of the 10 most recently active users, it is okay to serve them the usual JSON data and reset their count at 1.

Where to Work

We have five servers dedicated for COMP 112. Inside the departmental firewall (i.e. from a machine in Halligan, or from a public-facing server), please ssh to one of the following machines:

- comp112-01.cs.tufts.edu
- comp112-02.cs.tufts.edu
- comp112-03.cs.tufts.edu
- comp112-04.cs.tufts.edu
- comp112-05.cs.tufts.edu

You may only work on your server from these machines.

As you will learn in this course, every service runs on a port–a number between 1 and 32767. You will be assigned ports that are yours alone. To receive your port assignments, use the /comp/112/bin/ports command, which was written by Alva Couch.

```
~ > /comp/112/bin/ports
fdogar01, your ports are 9020-9039
PLEASE BE SURE TO WORK ONLY ON COMP112 MACHINES. THANKS!
~ >
```

When you test your services, you must stay within your own port range. Two services cannot share a port, so if you steal someone else's port, their service will behave strangely. Please be kind to your classmates, and do not steal their ports!

Testing

The servers on which you will write your code are protected from the outside world by a firewall. Your server will not be visible from the public internet. To test your server, you must do so from a point within the Halligan network.

You server will call functions from the network services library. When compiling, you will need to link against this library with the -lnsl flag:

```
gcc -g a1.c -lnsl
```

Start your program as follows, replacing 9020 with one of your port numbers:

```
./a.out 9020
```

At this point, your service is running. To see if it's working, open a browser on any Halligan machine and visit http://comp112-01.cs.tufts.edu:9020. (This is assuming you're on the comp112-01 server.) This should connect to your program and display the list of IPs that have accessed your page. Similarly, http://comp112-01.cs.tufts.edu:9020/about.html should display information about you.

If you are connecting remotely, you can also use the command-line wget and curl to test your server.

Submission and Lateness Policy

Submit your code by using provide: `provide comp112 a1 a1.c`

Late submissions will be accepted up to three days past the deadline, with a penalty of 10% per day late. For circumstances that you believe warrant an additional extension, speak to Prof. Dogar.

Useful Links

- http://www.linuxhowtos.org/C_C++/socket.htm
- http://www.cs.cmu.edu/~dga/15-441/S08/lectures/03-socket.pdf