

Programming Assignment 1

A Web Server

Due Saturday FEBRUARY 28, 2009 (11:59 pm)

1 Description

1.1 Part 1

The goal of this project is to build a functional HTTP/1.0 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.

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.purdue.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 Purdue computer science 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 `student/cs536/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 a 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. 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. Note that the previous discussion assumes the HTTP/1.0 protocol which is what you will be supporting in this first assignment.

Next, add simple HTTP/1.1 support to your web server, consisting of persistent connections and pipelining of client requests to your web browser. You will also need to add some heuristic to your web server to determine when it will close a “persistent” connection. 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. 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. That is, 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, it may not be able to afford to have an idle connection sitting around (consuming kernel/thread resources) for very long.

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/1.0 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 three main choices in how you structure your web server in the context of the above simple structure:

- 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 is largely appropriate because of its portability (relative to assuming the presence of a given threads package across multiple hardware/software platform). It does face increased context-switch 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 to the server (performing appropriate bookkeeping if so), and then it will loop overall 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.

You may choose from 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()`. We outline a number of resources below with additional information on these system calls. A good book is also available on this topic.

The format of the command should be:

```
myhttpd [<http>] [<port>]
```

If `< http >` is not passed, the server will run in HTTP/1.0 mode. Otherwise, passing 1 or 1.1 will define the HTTP version the web server will run in. 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

Now that you have a functional web server, the second part of the project 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 under various levels of offered load. 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 might be multi-threaded, with a different number of threads attempting to retrieve files as quickly as possible, or you may use a multi-process approach, where individual instances of your load generator start up, retrieve a file, and shut down. You can control offered load by controlling the number of simultaneous threads/processes that are retrieving files from the server. You should measure how long each request takes and keep track of this information for summary reporting (this will be used to generate average latency). You should also measure the total number of requests satisfied in a given time period (this will be used to generate average throughput).

A principal aspect of this assignment is to compare the performance of your web server using HTTP/1.0 versus HTTP/1.1, especially given the behavior of TCP. There is significant overhead to establishing and tearing down TCP connections (though this is less noticeable in a LAN setting) and persistent connections avoids this issue to some extent.

Can you think of any instances where 1.1 will underperform 1.0? A number of considerations affect the performance HTTP/1.0 versus 1.1. Principally consider some of the pros and cons of using a connection per session versus using a connection per object. Simply put, the difference between the two comes down to the following:

- Only a single connection is established for all retrieved objects, meaning that slow start is only incurred once (assuming that the pipeline is kept full) and that the overhead of establishing and tearing down a TCP connection is also only incurred once.
- However, all objects must be retrieved in serial in HTTP/1.1 meaning that some of the benefits of parallelism are lost.

Under what conditions would HTTP/1.1 underperform 1.0? Also 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).

It will be important to run your web server and load generator on different machines. It may be necessary to use multiple machines as load generators to saturate the server. You will want to keep track of CPU load on both the load generator and the server machines to determine when each system component saturates (at least with respect to CPU load).

Profile your server code to get a rough idea of the relative cost of different aspects of server performance. For different sized files, how much time is spent in establishing/tearing down connections versus transmitting files? Use timing calls around important portions of your code to keep track of this time on a per-logical operation basis. What type of bandwidth does the web server deliver as a function of file size? Also try running your test for different size files. How does latency/throughput change with file size? You may find it useful to use a scripting language, such as perl, to control the performance evaluation.

For extra credit, conduct your performance evaluation using different structures for your HTTP server. For example, if you decided to build your server using a multi-threaded model, implement an alternate

version that uses an event-based/single threaded approach for handling requests. How does the performance characteristics of your server change with different system structure?

2 Report

A very important aspect of your assignment will be your project report describing your system architecture, implementation, and high level design decisions. For your performance evaluation, you should include a number of graphs describing the various aspects of system performance outlined above. Make sure to clearly describe how you set up your experiments. What kinds of machines did you run on? What kind of network interconnected the machines? How did you build your load generator and collect statistics? How many times did you run each experiment to ensure statistical significance? Ideally, you will include error bars to indicate standard deviation or 95 % confidence levels. Finally, your writeup should include explicit instructions on how to install, compile, and execute your code.

Your writeups are due with the assignments at the times specified above.

3 Code and 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 Xinu Lab and 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* file describing how to compile and run your program from a terminal. Submit your project files in one ZIP file (including your report, more on the report in the following section) to cgaspard@purdue.edu by the due date. For any concerns about this assignment, please send an email to TAs Camille Gaspard (cgaspard@purdue.edu) or Zhen Zhu (zzhu@purdue.edu).

4 Grading Criteria and Demo

Please refer to the course web site for grading criteria. Students will demonstrate their projects in one of the two PSOs the week next to the due date of the assignment. Slots can be reserved during the PSOs of the week before the demonstrations week. For special arrangements please email cgaspard@purdue.edu. Students should prepare a 5-10 minutes demo based on the grading criteria available on the course web site.