CS 536: Data Communications and Computer Networks Fall 2013

Programming assignment 1: Building and evaluating a web server

Due: Thursday, September 12th, 2013 (by 11:59 PM) -- Hard Deadline

1. Description

1.1 Part 1: Web Server

The goal of this assignment is to build a functional HTTP server. The 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 must work *individually* on this assignment; we will use MOSS (http://theory.stanford.edu/~aiken/moss/) to check for plagiarism.

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 file system. For example, you may 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 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 which is what you will be supporting first.

Next, add simple HTTP/1.1 support to your web server: support persistent connections and pipelining of client requests. You will need to add a 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 configurable in the server*. It can be dynamically adjusted 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 somewhat 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 you 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 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 can refer to the HTTP/1.1 RFC to learn more about the method syntax and error codes. For this assignment, only the GET command is required (you need not implement HEAD, PUT, DELETE, TRACE, OPTIONS, etc). You can check the validity of your implementation by issuing queries from a telnet session or from the client described in next section.

You may choose from C or C++ (other languages like Java or C# are not allowed). You will want to become familiar with the interactions of the following system calls to build your system: socket(), select(), listen(), accept(), connect(). See section 5 at the end of this document for good tutorials on sockets, as well as threads, etc.

The format of the command should be:

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

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.

<timeout> is given as a positive time in seconds. If <timeout> is not passed, your web server should default the timeout used for HTTP/1.1 which is 300 seconds (5 minutes). If <timeout> is given, your web server should perform basic error checking to ensure it is greater than 0 seconds.

Do not include the brackets on the command line or in your command-line parsing code (the angled brackets are just for clarity and the square brackets indicate the values are optional). Running your web server should resemble the following:

```
myhttpd 1.1 65000 20
```

This would run the web server in HTTP/1.1 mode, listening on port 65000, and have a timeout of 20 seconds.

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 under various levels of offered load. You will measure server performance in terms of throughput (requests/second) 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).

1.3 Part 3: Use Mininet to Run Experiments and Compare Two Different Protocols

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.

As discussed in Chapter 2 of our textbook, there is significant overhead to establishing and tearing down TCP connections (though this is less noticeable in LAN settings with short propagation delays) and persistent connections avoid this issue to some extent.

A number of considerations affect the performance HTTP/1.0 versus 1.1. Consider primarily 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.

Consider the effects of (1) the delay-bandwidth product (i.e., bandwidths and round trip times), and (2) 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).

You can emulate different scenarios by using Mininet to set up a virtual network, where you can configure the topology, bandwidth and delay. The following steps are recommended for installing Mininet:

- Download a Mininet VM image
- Download and install a virtualization system. VirtualBox is recommended since it is free and available for OS X, Windows, and Linux. You can also use Qemu, VMware, or KVM as alternatives.
- Open the Mininet VM image using your virtualization system (e.g., VirtualBox)
- You can work on VirtualBox directly. However, it is recommended that you set up an SSH into Mininet by following these setup notes.

Please post to the course page on Piazza if you have any question on installing or using this tool.

It is important to set up multiple load generators to saturate the server. You may want to keep track of CPU load on your machine to determine when the 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 and delay-bandwidth products, 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 throughput does the web server deliver as a function of file size and delay-bandwidth product?

You may find it useful to use a scripting language, such as PERL, to control the performance evaluation so that you do not have to manually repeat a large number of experiments.

You will find the following paper helpful:
Barford, Paul, and Crovella, Mark E. A Performance Evaluation of Hyper Text Transfer Protocols, In Proceedings of ACM SIGMETRICS, May 1999, pages 188-197.

2. Report

An important component of your assignment will be your 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. For example, what kinds of networks did you set up on Mininet? How did you build your load generator and collect statistics? How many times did you run each experiment to ensure statistical significance? Ideally, you should 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 report is also due with the assignment at the date/time specified above.

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 XINU Lab (HAAS 257)* 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. Submit your project files, including your report, using *turnin tool*, by the due date/time.

Here is how to use *turnin*

- Log in to a XINU lab machine, e.g., xinu1.cs.purdue.edu
- Type "*turnin -c cs536 -p pa1 <name of directory with your files>*" from the parent directory of the directory with your files.
- You can verify your submission by typing the following command:

  turnin -v -c cs536 -p pa1

  Do not forget the -v above, as otherwise your earlier submission will be erased (it is overwritten by a blank submission).
  Note that resubmitting overwrites any earlier submission and erases any record of the date/time of any such earlier submission.

For more information on the *turnin* tool, run *man turnin*. If you have concerns or questions, please feel free to send email to the TA.

4. Grading Criteria and Demo

The grading criteria for this programming assignment are given at here. Students will demonstrate their projects in one of the two PSOs after 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 the TA. Students should prepare a 5-10 minutes demo based on the grading criteria linked above.

5. Resources

The links to the resources can be accessed by clicking here.