Project - A Simple Web Server

EECS-345 - Winter 2010

Important Dates

Out: January 15, 2010.

Due: February 10, 2010 (midnight).

Project Overview

For this project you are asked to develop a functional HTTP/1.0 server. This project should be done in teams of two.

Educational Objectives

The project will teach you the basics of distributed programming, client/server structures, and some of the many issues in building high performance servers.

Background

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.

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 `~fabianb/swslab/server/files/`, which we call the document root. When your server gets a request for index.html , 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 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. Provide simple server support for ".htaccess" files on a per-directory basis to limit the domains that are allowed access to a given directory. You only need to implement the `allow/deny from 000.000.000.000/all` syntax and rules should be applied in descending order. You should be able to allow/deny from both ip addresses as well as domain names.

Remember also 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.

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 initially supporting.

Once you are done, you will 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, 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 or IE) to connect to your web server and retrieve the contents of the course front page from it. 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 to 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 either Java, C pr C++ to build your web server but you must do it in a Unix-like environment. You will want to become familiar with the interactions of the following system calls to build your system: `socket()`, `select()`, `listen()`, `accept()`, `connect()`. I will list a number of useful resources in the course page.

Deliverables and hand-in instructions

You must include a makefile with your submission. When we run make along with your makefile the webserver should be created as a single file called server.

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`. Thus, we should be able to run your webserver as

```
$ ./server -document_root "/home/fabianb/swslab_files"
```

Note that there is no slash at the end of `swslab_files`.

Make the port the server listens on, a command line option. The option must be specified as -port . Thus, we should be able to run your server as

```
$ ./server -document_root "/home/fabianb/swslab_files" -port 8080
```

To submit your assignment, email a tarball of your assignment to eecs-345-ta@cs.northwestern.edu with the subject line "SWS Project".

Please include the following files in your tar ball.

The deliverables for this project include:

1. A short writeup that lists the names of the people involved, what functionality is implemented in your webserver and what, if any, problems it has.
2. All the files for your source code only. Please do not include any executables.
3. The makefile.

Good luck!