csci-e28
Assignment 6: Web Server Next Generation

Introduction

The World Wide Web is a set of files and directories scattered across the Internet. These files and directories are linked to each other with 'hyperlinks.' Hyperlinks are references to other, related documents. A hyperlink is a string of the form: `http://machine-name:portnumber/path-to-item`.

Users 'visit' sites. That is, they look at lists of documents, and they click on the name of a document to view it. Some documents contain text, some contain pictures, some contain sound clips, etc. Despite all the glitz and animation, the web is a set of directories and files on millions of machines.

The software that makes the web possible consists of web servers and web clients (also known as web browsers.) For this project, you will add several features to the web server discussed in class. In the process, you will learn how the web works, and you will be able use and review many of the ideas and skills you have learned in this course.

How the Web Works: ls, cat, exec

You have files, directories, and programs on your machine, and you want to make some of them available to people on other machines. You could give them accounts on your machine, and they could use those accounts to login, read files, list directories, and run programs.

What if you do not want to give them accounts, or what if you cannot give them accounts? Managing accounts takes work and can open your machine to potential dangers. If you do not run the machine, your system administrator may not want to give out accounts to your pals.

The web solves the problem of making files, directories, and programs available to people on other machines without giving them accounts. Here's how it works. You run a server program that knows how to list directories, cat files, and run programs. That program is called a web server. Users run programs called web clients, also known as web browsers, that know how to ask the server program to list directories, display file contents, and run programs. The server only shows the clients the directories and files within specified directories.

This method has lots of advantages. Nobody needs an account, nobody needs a password. Readers only have access to directories and files you want them to see. Nobody logs in, so you have fewer security problems.

To summarize, a web server is a program that runs on your machine and provides access to a specific set of files and directories on your machine. Remote users can connect to the server and ask the server to list a directory, get the contents of a file, or run a program. The server is a program that performs the functions of `ls`, `cat`, can `exec` for remote users.

How the Client and Server Communicate: HTTP

In class we looked at some client/server programs, and we examined the hypertext transfer protocol. In all those cases, a server program sets up a socket and waits for a call to come in. When a call comes it, it accepts the call and talks with the client program through a socket. The 'conversation' is done with reads and writes to the socket. It is a lot like two people talking through a telephone connection.

Here is a simple dialog between a web client and a web server. It consists of two parts: the client asks for a file or directory, the server sends back a response.

```
client says              server says                            explanation
-------------------------------------------------------------------------------
GET mars.txt HTTP/1.0 ->                                        client asks for a file
                      ->                                        blank line ends request
                      <- HTTP/1.1 200 OK                        server says OK
                      <- Date: Sun, 19 Apr 1998 06:27:44 GMT    server sends some info
                      <- Server: Apache/1.2.4                   about itself
                      <- Connection: close                      about the connection
                      <- Content-Type: text/plain               about the file
                      <-                                        blank line ends header
                      <- Martians Land in New Jersey!           file contents start here
                      <- --------------------------             and continue
                      <- This evening, Martian pods landed      until
                      <-  ...                                   done
```

The server sets up its service at a port on a machine. It waits until a call comes in. The client connects to that service. The server accepts the call and reads input. The client sends its request. In the example shown above, the request consists of one line:

```
GET mars.txt HTTP/1.0
```

This is a request from the client to GET a particular file. The client sends a single blank line to indicate the end of its request.

After the client sends that blank line, it is the server's turn to talk and the client's turn to listen. The server first sends a status report. In the example above, the status report is

```
HTTP/1.0 200 OK
```

which means 'ok, we have that file and shall send it to you.' Each type of report has a numerical code. The code for 'OK' is 200. The code for 'we don't have that file' is 404. There are several codes defined. The documentation explains them all.

After the single status report line, the server sends zero or more lines of additional information. The example above shows that the server sends back the date, the server's name and version, the type of connection, and the type of file. The status line and additional information comprise the response header. A single blank line indicates the end of the header.

After that single blank line, the server sends the contents of the file or the listing of the directory the client requested. After sending the file contents or the directory listing, the server closes the socket. The client sees EOF and shows the stuff to the user.

That is a typical transaction between web client and web server. The client sends a request using a special command format. The server then responds to the request using a special response format. The format of these requests and responses is called HTTP - hypertext transfer protocol.

The details of HTTP are described in the file called RFC1945.txt . A copy of this file is in the assignment directory. Now that you know the basic idea, you should be able to make decent sense of the RFC.

What the Server Does: Read a Request, Send back a Reply

The logic in a server is pretty simple. It accepts a connection from a remote client. It reads from the socket to see what file or directory the client wants to see. When the client is done with its request, the server gets the file or directory listing and sends it back with an appropriate header. The server then hangs up the connection and waits for the next call to come in.

This web server is sort of like the shell you wrote. The shell read a line of input, looked at the line to figure out what the user wants, performed the requested operation, then went back and waited for the next command.

What Requests Can a Client Make? : GET, HEAD, POST

Most of all http transactions involve the GET, HEAD, and POST requests.

GET — The GET method tells the server to send back the contents of the named file or a listing of the named directory. The syntax is extremely simple. The request consists of three strings on one line. The first string is the word GET, the second string is the name of the file or directory, and the third string is the version of HTTP the client speaks. Note: if the client is requesting a directory, the directory name *must* end with a slash: `/` .

HEAD — The HEAD method tells the server to send back just the header, not the contents, for the named file or directory. This request consists of three strings on one line:

```
HEAD  name-of-file-or-directory http-version
```

Why is this useful? There are lots of reasons. The client can use this method to see if the file is there and perhaps find out some information about the file or directory.

POST — The POST method is the opposite of GET. It allows the client to send data to the server. It was originally created to allow users to send news articles or pieces of text to the server.

Your web server only has to support the GET and HEAD requests.

Limiting Access: A Virtual Root

You have many files and programs, but you only want to make some of them available to people on other machines. Web servers make this easy. You create a subdirectory in your account and make sure the web server only shows people files in that directory and directories below it.

The filenames passed to the server with the GET, HEAD, and POST requests are all taken relative to the top of this subdirectory tree. If a user requests the listing of the directory called `/hw/`, the server removes the leading slash and opens the directory called `hw/` . If the user requests the file called `/hw/wsng/RFC1945.txt` the server sends back the contents of the file called `hw/wsng/RFC1945.txt`.

If a user asks for the listing of a directory called `../../` the server, as Unix does at the top of its directory tree, takes `/..` to be the same as `/`.

What Your Server Has to Do: A General Description

The web server you are given actually does all three things: lists directories, displays files, and runs programs, but it does a minimal job of each task. Your job is to understand the structure of the code and enhance each of these operations.

Getting Started

[1] Login to your account on ice and make a subdirectory for the project.

[2] Change into your project directory and type `~lib215/hw/wsng/setup.wsng`

The setup script will copy the files you need, and it will create a sample web site in your account. If you already have a web site, it will not damage what you have.

[3] Your website is just a directory in your account. The directory is under your home directory. It is called `public_html/wsng` .

Use cd, ls, and cat to look at your web site.

[4] Test your website by browsing it with any web browser. Use as the address:

`http://www.people.fas.harvard.edu/~`*your-logname*`/wsng/`

You should see the same files and directories you saw with cd and ls. Click on some of the things to see what is on the site. It should contain the same stuff you saw with ls and cat.

The web server that is responding to the requests from your browser is the web server run by the Science Center computer services.

[5] Now compile and run your web server.

In the directory for this project, type `make` It will compile a custom version of **ws**.

Then run your web server by typing `./ws` .

The server will tell you the port number it is using and the machine it is running on.

[6] You may now browse your site using the hostname and port number. The address is:

`http://`*machine*`.fas.harvard.edu:`*portnum*`/`

IMPORTANT: On the machines we use, running a server on a non-standard port is blocked. To connect to your server, you have to set up a *tunnel* to get past the blocking. See the section at the end of this document for details.

When you connect (directly if allowed, or through a tunnel), you will see a directory listing of your website. This time, though, the directory listings are being sent back by the web server you just compiled and are now running.

Notice how the web server sends to standard output copies of the requests it receives. You can see the headers the web browser is sending after its one-line request.

Try the following URLs:

`http://`*machine*`.fas.harvard.edu:`*portnum*`/file1.html`
`http://`*machine*`.fas.harvard.edu:`*portnum*`/dir1`
`http://`*machine*`.fas.harvard.edu:`*portnum*`/file2.txt`
`http://`*machine*`.fas.harvard.edu:`*portnum*`/file3.jpg`

Notice how tedious it is to type these URL's into the browser. When you use a regular server, the directory listing has links. Why can't your server put links on the listing?

[6] Look through the http description in rfc1945. It is a long document, but you already know many of the basic ideas. Look at the return codes and the sorts of header information servers and clients exchange.

[7] Look at the code for wsmk.c . We discussed the operation and structure of the program during class. The code has comments and is pretty modular. The program handles files, directories, and programs. A program is designated by an extension of `.cgi` .

Your Assignment

Make a copy of ws.c (or start from scratch if you are ambitious), call the copy wsng.c, and modify the program so it meets the following specifications:

1. Better Reply Headers

   The current version sends back only the status line and the file type. It does not send back a date, a server name and version, nor any of the other goodies. Your program must send back a date and a server name and version. You may add any other items you like. See the RFC for options.

2. Support the HEAD method

   Modify the program so it handles requests from the client for the HEAD for a document. The CGI

specification (rfc3875) states:

> The HEAD method requests the script to do sufficient processing to return the response header fields, without providing a response message-body. The script MUST NOT provide a response message-body for a HEAD request. If it does, then the server MUST discard the message-body when reading the response from the script.

3. Table-Driven File Types

   The server is coded to handle files with extensions of .html, .jpg, .jpeg, and .gif . All other file types are sent back with a default file type of *text/plain* . This approach is not flexible. Modify your program so it uses a table-driven system to associate *Content-type* with file extensions. The starting version of ws.c reads a configuration file called wsng.conf for the path to the site and the port. Examine this file. Extend this configuration file and the code to process the file so it can handle lines of the form:

   ```
   type jpg   image/jpeg
   ```

   The configuration file must contain, and your code must handle a line of the form:

   ```
   type DEFAULT text/plain
   ```

4. Directory Listings with Links

   Browse your site with the regular web server. The directory listings have clickable links. The directory listings produced by ws.c do not. Browse your site with the regular server and view the source of a directory listing. Notice how the directory listing contains links to the files in the directory. Your program must produce directory listings with links.

   This is not too difficult, but it requires you change the way wsng.c lists directories. Simply using exec to run `ls` does not include the links. You need to devise a different method. There are several. Be sure you still display the file size and modification time.

5. index.html, index.cgi

   Your sample site contains a directory called dir1. When you ask to GET this directory with a regular server, you get the contents of the file called index.html which is in that directory. When you ask the wsng to view the contents of dir1, you just get a list of files.

   The rule for GETing a directory is as follows:

   a. If the directory does not exist or is not readable report an error
   b. If the directory contains a file called `index.html` send back its contents with Content-type: text/html
   c. If the directory contains an executable file called `index.cgi` run that program and send back the output of that program
   d. If the directory contains neither `index.html` nor `index.cgi` send back the actual directory listing.

   Your server has to implement this rule.

5. Handles Errors

   The server handles several types of errors. Read the RFC to see which ones the server you are given does not handle. Add one more error case.

6. Zombies

   The server takes the call, creates a new process to handle the request, then loops back to take the next call. Notice the parent does not call wait() the way the shell does. Why not?

   The problem with not calling wait() is that the child processes become zombies. If your shell receives hundreds of hits, it will create hundreds of child processes, and they all will turn into zombies. Read the manual, the texts, or the web to find a solution to the zombie problem. There are several ways to solve it.

7. QUERY_STRING

Web servers list directories, display files, and run programs. There are several systems servers use to designate which files are program. Our server uses the filename extension system. If a filename ends with the extension `.cgi`, the server runs the program. The original server `ws.c` should be able to run the cgi programs in the sample site. For example:

`http://`*machine*`.fas.harvard.edu:`*portnum*`/dir2/hello.cgi`

should work fine. Make some other cgi programs and try them.

This feature is not complete, though. A real web server accepts requests like

`http://`*machine*`.fas.harvard.edu:`*portnum*`/dir2/showcal.cgi?3`

which is presented to the server as:

```
GET /dir2/showcal.cgi?3 HTTP/1.0
```

Notice how the argument to the GET command includes a question mark. The string after the question mark is called the *query string*. The server handles arguments of this form by doing the following:

a. split the argument at the question mark
b. the string before the question mark is the regular argument
c. the string after the question mark is stored in the environment as the value of the variable `QUERY_STRING` .
d. the environment variable `REQUEST_METHOD` is set to "GET"
e. The cgi program is run, inheriting the `QUERY_STRING` and `REQUEST_METHOD` variables from the server.

Add this behavior to your server. You should now be able to visit the address

`http://`*machine*`.fas.harvard.edu:`*portnum*`/dir2/showcal.cgi?3`

As you add these new features to your server, you can run the server and test it by browsing your site with any browser. By the time you have all the sections done, you should be able to visit all parts of your website. All the directories, files, and cgi programs should work.

Testing and Submitting Your Project

Check the course website for information about testing your program. There may be a test script. Test your server with a browser other than Internet Explorer. That browser does not follow the RFC in the way it handles content type.

Submit the usual Makefile, Plan, README, and source code with the command `~lib215/handin wsng` .

Appendix: Setting up and Using an SSH Tunnel

Using an SSH Tunnel to Connect to Your Server

1. The Problem

Servers at FAS block connections to non-standard ports from off-campus machines. Your web server listens on a port number equal to your user id number, not a commonly-used port. Thus you cannot connect directly to your server if you are working from home or some other off-campus location.

2. The Solution: An SSH Tunnel

The solution is pretty simple: create a tunnel that connects your home machine to the web server running on nice.

First, what is a tunnel? Consider, for example, the Ted Williams Tunnel in Boston. It has one end in South

Boston and the other end in East Boston. The harbor prevents cars from going directly from South Boston to East Boston, but cars can enter one end of the tunnel and arrive dry and safe at the other end.

Now, what is a network tunnel? A firewall setting at the fas machines prevents connections to non-standard ports, but you can create a tunnel with two ends -- one end on your local machine and the other end at nice. By connecting to a port on your local machine, you can get routed to the corresponding port on the remote machine without being blocked.

3. All Users - Linux, OSX, Windows

The following steps apply to all users.

1. Login to your account on nice
2. type: `id`
3. Write down the value of your uid (that's user id)

4. Linux and Mac OSX Users

The software to create ssh tunnels is already installed on GNU/Linux and OSX machines. For the purposes of this example, we shall represent your uid with UUUU .

1. On your local machine, in a terminal window, type:
   `ssh -L UUUU:localhost:UUUU yourname@nice.fas.harvard.edu`
2. Start your web server:
   `./ws`
3. On your local machine, browse your server using URL:
   `http://localhost:UUUU/`

That's it. The tunnel has created a server on your local machine listening at port UUUU. Any connection that server receives is then sent through your regular ssh connection to the ssh server on nice. That server then becomes a client to your web server. But your browser does not know the server at localhost is just transferring data to the *real* server elsewhere.

5. Windows Users

The instructions are almost exactly the same as those in the preceding section. The only difference is that you need to get a piece of software to do what ssh does.

1. Use google to search for puTTY
2. Go to the puTTY download site (at chiark.greenend.org)
3. Get plink.exe OR the full installation package
4. Install what you get

To use plink, just follow the instructions in the previous section replacing the command *ssh* with the command *plink* .