Lab 5_ Building a HTTP Server

Updated automatically every 5 minutes

Lab 5: Building a Concurrent HTTP Server

CS252

Frequently asked questions

Lab 5 Slides

Your server should look like this: This is the http-root-dir as served by the CS HTTP Server.

Grading Form

Pre-reading for this Lab

Before coming to the lab, study carefully the example server given in [1]. Also, familiarize yourself with the following functions in the socket API:

Lab 5_ Building a HTTP Server

Updated automatically every 5 minutes

The objective of this lab is to implement a HTTP server that will allow a HTTP client (a web browser like FireFox or Internet Explorer) to connect to it and download files.

HTTP Protocol Overview

A HTTP client issues a `GET' request to a server in order to retrieve a file. The general syntax of such a request is given below :
GET <sp> <Document Requested> <sp> HTTP/1.0 <crlf>
{<Other Header Information> <crlf>}*
<crlf>
where :

- <sp> stands for a whitespace character and,
- <crlf> stands for a carriage return-linefeed pair. i.e. a carriage return (ascii character 13) followed by a linefeed (ascii character 10).
- `<crlf><crlf> is also represented as "\r\n\r\n".`
- <Document Requested> gives us the name of the file requested by the client. As mentioned in the previous lab, this could be just a backslash (/) if the client is requesting the default file on the server.
- {<Other Header Information> <crlf>}* contains useful (but not critical) information sent by a client. Note that this part can be composed of several lines each separated by a <crlf>.
- `* - kleene star // regular expressions`

Finally, observe that the client ends the request with two **carriage return linefeed character** pair: <crlf><crlf>
The function of a HTTP server is to parse the above request from a client, identify the file being requested and send the file across to the client. However, before sending the actual

Published by Google Drive – Report Abuse

<sp> follows <crlf>
Server: <sp> <Server-Type> <crlf>
Content-type: <sp> <Document-Type>
<crlf>
{<Other Header Information> <crlf>}*
<crlf>
<Document Data>

where :

- <Server-Type> identifies the manufacturer/version of the server. For this lab, you can set this to CS 252 lab5.
- <Document-Type> indicates to the client, the type of document being sent. This should be "text/html" for an html document, "image/gif" for a gif file, "text/plain" for plain text, etc.
- {<Other Header Information> <crlf>}* as before, contains some additional useful header information for the client to use.
- <Document Data> is the actual document requested. Observe that this is separated from the response headers be two carriage return - line-feed pairs.

If the requested file cannot be found on the server, the server must send a response header indicating the error. The following shows a typical response:
HTTP/1.1 <sp> 404 File Not Found
<crlf>
Server: <sp> <Server-Type> <crlf>
Content-type: <sp> <Document-Type>
<crlf>
<crlf>
<Error Message>
where :

- <Document-Type> indicates the type of document (i.e. error message in this case) being sent. Since you are going to send a plain text message, this should be set to text/plain.
- <Error Message> is a human readable description of the error in plain text/html format indicating the error (e.g. Could not find the specified URL. The server returned an error).

Algorithm Details

Getting started

Login to a CS department machine (a lab machine or `data.cs.purdue.edu`), navigate to your preferred directory, and run

```
cd
cd cs252
git clone
/homes/cs252/sourcecontrol/work/$USER/lab5-
src.git
cd lab5-src
```

Then build the server by typing *make*. Run the server by typing *daytime-server* without arguments to get information about how to use the server. Run the server and read the sources to see how it is implemented. Some of the functionality of the HTTP server that you will implement is already available in this server.

Basic Server

You will implement an iterative HTTP server that implements the following basic algorithm:

- Open Passive Socket.
- Do Forever
    - Accept new TCP connection
    - Read request from TCP connection and parse it.
    - Frame the appropriate response header depending on whether the URL requested is found on the server or not.
    - Write the response

that is found in htdocs/index.html) to TCP connection.
○ Close TCP connection

The server that you will implement at this stage will not be concurrent, i.e., it will not serve more than one client at a time (it queues the remaining requests while processing each request). You can base your implementation on the example server given in [1] . The server should work as specified in the overview above. Use daytime server as a reference for programming with sockets. Implement http server in "**myhttpd.cc**".

Basic HTTP Authentication

Background

In this part, you will add basic HTTP authentication to your server. Your HTTP server may have some bugs and may expose security problems. You don't want to expose this to the open Internet. One way to minimize this security risk is to implement basic HTTP authentication. You will implement the authentication scheme in RFC7617, aptly called Basic HTTP Authentication.

In Basic HTTP Authentication, you will check for an `Authorization` header field in all HTTP requests. If the `Authorization` header field isn't present, you respond with a status code of `401 Unauthorized` with the following additional field: `WWW-Authenticate: Basic realm="<something>"` (you should change `<something>` to a realm ID of your choosing such as "The_Great_Realm_of_CS252"). When your browser receives this response, it knows to prompt you for a username and password. Your browser will encode this in Base64 in the following format: `username:password` that gets supplied in the `Authorization` header field. Your browser will repeat the

`mycredentials.txt | base64`.
Client Request:

```
      GET /index.html
      HTTP/1.1
```

Server Response:

```
      HTTP/1.1 401
      Unauthorized
      WWW-Authenticate:
      Basic
      realm="myhttpd-
      cs252"
```

Client browser prompts for username/password. User supplies `cs252` as the username and `password` as the password and the client encodes in the format of cs252:password in base 64.
Client Request:

```
      GET /index.html
      HTTP/1.1
      Authorization:
      Basic
      Y3MyNTI6cGFzc3dvcmQ=
```

Note: you should create your own username and password and NOT use cs252 and password. You shouldn't use your Purdue career account credentials either.

You will check that the request includes the line

"Authorization: Basic <User-password in base 64>"

and then respond. If the request does not include this line you will return an error. This Line will be included in all the subsequent requests so you do not need to type user password again.

After you add the basic HTTP authentication, you may serve other documents besides index.html.

RFC7617

Adding Concurrency

You will also add concurrency to the server. You will implement three concurrency modes. The concurrency mode will be passed as argument. The concurrency modes you will implement are the following:

-f : Create a new process for each request

In this mode your HTTP server will fork a child process when a request arrives. The child process will process this request while the parent process will wait for another incoming request. You will also have to prevent the accumulation of inactive zombie processes. You can base your implementation on the server given in [3]

-t : Create a new thread for each request

In this mode your HTTP server will create a new thread to process each request that arrives. The thread will go away when the request is completed.

-p: Pool of threads

In this mode your server will put first the master socket in listen mode and then it will create a pool of 5 threads where each thread will execute a procedure that has a while loop running forever which calls accept() and dispatches the request. The idea is to have an iterative server running in each thread. Having multiple threads calling accept() at the same time will work but it creates some overhead under Solaris (See [4]). To avoid

Lab 5_ Building a HTTP Server

Updated automatically every 5 minutes

The format of the command should be:

myhttpd [-f|-t|-p] [<port>]

If no flags are passed the server will be an iterative server like in the Basic Server section. **If <port> is not passed, you will choose your own default port number**. Make sure it is larger than **1024** and less than **65536**.

MAKE SURE THAT THERE IS A HELP FUNCTION AND YOUR CODE IS INDENTED AND EASY TO READ (there will be points for this).

Turn in the project before Monday November 30th 11:59pm

1. *Login to a CS department machine*
2. *Navigate to your `lab5-src` directory*
3. *Run `make clean`*
4. *Run `make` to check that your shell builds correctly*
5. *Run `git tag -f part1`*
6. *Run `git push -f origin part1`*
7. *Run `git show part1`*
8. *The `show` command should show the diff from the most recent commit*

Published by Google Drive – Report Abuse