

Class: CSE 130
Professor: Faisal Nawab
Assignment 1
Fall 2020

Design Document: httpserver.cpp

Contributors:

Adriel Naranjo - adjnaran
Abbas Engineer - afengine

Objective & Synopsis

To implement a simple single-threaded HTTP server. The server will respond to simple GET and PUT commands to read and write “files” named by 10-character ASCII names. The server will persistently store files in a directory on the server, so it can be restarted or otherwise run on a directory that already has files.

This project requires the one constructing the httpserver to have an understanding of various web technologies.

In this project we utilize the tools provided by C/C++ libraries which are [listed below](#) and should be looked into before considering the rest of the design. Emphasis on the following:

`<sys/socket.h>`, `<netdb.h>`, `<netinet/in.h>`, `<errno.h>`, `<sys/stat.h>`
`<stdio.h>`, `<stdlib.h>`, `<sys/types.h>`, `<fcntl.h>`, `<unistd.h>`, `<string.h>`,
`<unistd.h>`, `<iostream>`, and `<ctype.h>`

Initial Assumptions:

1. Work Around not using FILE * Library
2. Send and receive files via http - The client sends a request to the server that either asks to send a file from client to server (PUT) or fetch a file from server to client (GET).
3. The Server binary is called **httpserver**
4. Resource names must be 10 ASCII characters long - Must consist only of the upper and lowercase letters of the (English) alphabet (52 characters), and the digits 0–9 (10 characters).
5. The server must respond to a PUT or GET with a “response” - Example Response Header: HTTP/1.1 200 OK\r\nContent-Length: 0\r\n\r\n
6. The Server should use the directory in which it's run to write(2) files that are PUT, and read(2) files for which a GET request is made. All file I/O for user data must be done via read() and write()
7. Need to use the system calls: socket, bind, listen, accept, connect, send, recv, open, read, write, close.

Let's get started:

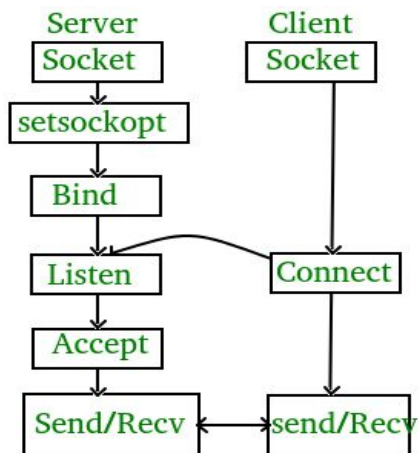


Diagram Citation: <https://www.geeksforgeeks.org/socket-programming-cc/>

This diagram from Geeks helped us initially understand the basic design of this project. To create a client-server model in C/C++ there 3 major steps that need to be taken:

Step 1: create a root socket that can listen for incoming connections.

Get the information needed in order to create a host socket. A lot of references of this step can be found at: <https://man7.org/linux/man-pages/man3/getaddrinfo.3.html>. Other functions seen in this step can also be found in their own manpage(s).

```
root_socket_init()
    case 1:
        preferred_port = argv[2];
        check_alphabet(preferred_port); // just checks if the
port is all digits 0-9
        case 2: preferred_port = "80";

    // Check the manpages for these, understanding these structures will
help you
    // create various types of sockets.
    struct addrinfo *addrs, hints;
    getaddrinfo(hostname, preferred_port, &hints, &addrs);
    // initialize the root socket using socket()
    root_socket = socket(AF_INET, SOCK_STREAM, 0);
    Set_socket_attributes(root_socket); // see link at beginning of step 1
to see how to do
    //After validating the socket, put the socket to use by listening.
    bind(root_socket, address);
    listen(root_socket);
```

Step 2: accept the connection and interpret the request from the new connection (client) to the root.

```
// keeps the program state to listening so incoming client requests may
come in
while(1){
    // where is address set using getaddrinfo & set_socket_attr
    New_connection = accept(root_socket, address);
    // read incoming message
    read(new_connection, buffer)
    // check manpages, sscanf extracts info from formatted messages
from http
    sscanf(buffer, "%s %s %s", request, file, protocol);
```

Step 3: from the server side (server), process the request

Due to the restrictions of the file name we must first validate the file resources that meet the constraints of this project.

If the file is longer than 10 characters long:

Response 400 Bad request

If the file contains any special characters, non-alphabet, non-numeric

Response 400 Bad request

Next, since we know we have a valid file resource name we must check if the request type is either a PUT or a GET. Once this information is obtained we can continue responding depending on the context.

If GET:

Create buffer to read from file if opened successfully

get_file_info () // use stat() check manpages

open(file);

If permission to open file is not granted

Response forbidden 403

If file does not exist in server directory

Response not found 404

If file has permissions and exists

Create 200 OK Response and read file contents from the file

directory

into the buffer. Then write buffer contents to the socket to

give the client

the requested data.

If PUT:

Retrieve information about the content so we can use it to create the response.

```

        Create a buffer for data transfer.
        open(file); // this is the file that exists in the server to be
written
        recv(new_socket, buf) // information to be PUT into file is
received into the buffer
        write(fd, buf) // write the information into the file that exists
on the server
        If None:
            Response 400 Bad request

        close(new_connection)
} // NOTICE: this while loop start in step 2 and ended in step 3

```

Question:

What happens in your implementation if, during a PUT with a Content-Length, the connection was closed, ending the communication early? This extra concern was not present in your implementation of dog. Why not? Hint: this is an example of complexity being added by an extension of requirements (in this case, data transfer over a network).

Answer:

Initially with handling the PUT requests the connection was not being closed to begin with. Ideally I believe our client would store whatever information it could, then inform the sender that the connection was terminated, resend the request. This extra concern was not present in our dog implementation because there was no need for using sockets that act as the endpoints to manage the connection. With the httpserver there may be a queue of clients, whereas with the dog program, the code would interact directly with a client, and once all bits were received, the program would close. Frankly this is the major difference between the two. This assignment has many simple components but if not invoked properly, like leaving a connection open when it should be closed can have a huge effect on the interaction between client and server.

Error from our testing:

```

* transfer closed with 24 bytes remaining to read
* Closing connection 0
curl: (18) transfer closed with 24 bytes remaining to read

```

Testing Issues:

We had some issues at first with getting the right structure for our design. We did not understand that we needed to keep the listening state on and this was accomplished with the while(1) loop later.

Another issue we ran into as far as structure goes, is that we initially tried to compartmentalize everything into components but as the number of helper functions grew along with components, we started running into problems with our data. So we had to rewrite everything so that the

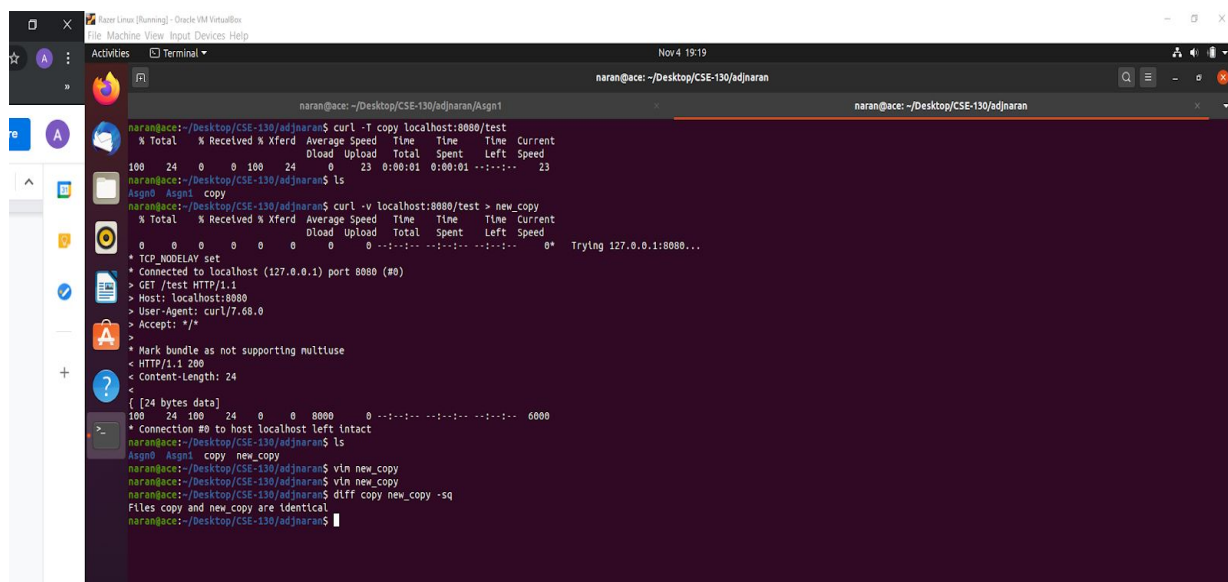
order of our program was executing correctly so that it wouldn't close the root socket unintentionally and not mess up the request made by the client from the new connection.

Lastly we had issues with understanding test cases. We would run curl commands that were identical in expected behavior but would create different effects when run against our httpserver.

Conclusion:

So this is how we tested it. Just for reference.

- 1) In the Assignment 1 directory, I created a file called 'test' which contains "This is test text from the server"
- 2) I ran the **httpserver** program and in a separate directory and I created a file called 'copy' which contains text: This is text from copy.
- 3) The first command I run will trigger a PUT: `$ curl -T copy localhost:8080/test`
- 4) Next I want to make sure the info was put in the server so then I run another command that will trigger a GET: `$ curl -v localhost:8080/test > new_copy`
- 5) Now that I received the data and placed the data into the file new_copy, I have two files that should be identical.. Copy (original) and new_copy (arrived data).
- 6) To ensure the data is the same in these files I then run: `$ diff copy new_copy -sq`
- 7) Tada! This is how our web server handles requests and it's pretty rad but was a pain in the butt!



```
naran@ace: ~/Desktop/CSE-130/adjnaran/Asgn1
naran@ace: ~/Desktop/CSE-130/adjnaran
naran@ace: ~/Desktop/CSE-130/adjnaran/Asgn1
naran@ace:~/Desktop/CSE-130/adjnaran$ curl -T copy localhost:8080/test
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 24 0 0 100 24 0 23 0:00:01 0:00:01 --:--:-- 23
naran@ace:~/Desktop/CSE-130/adjnaran$ ls
Asgn1 Asgn1 copy
naran@ace:~/Desktop/CSE-130/adjnaran$ curl -v localhost:8080/test > new_copy
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0* Trying 127.0.0.1:8080...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /test HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200
< Content-Length: 24
<
[ 24 bytes data]
100 24 100 24 0 0 8000 0 --:--:-- --:--:-- --:--:-- 6000
* Connection #0 to host localhost left intact
naran@ace:~/Desktop/CSE-130/adjnaran$ ls
Asgn1 Asgn1 copy new_copy
naran@ace:~/Desktop/CSE-130/adjnaran$ vln new_copy
naran@ace:~/Desktop/CSE-130/adjnaran$ vln new_copy
naran@ace:~/Desktop/CSE-130/adjnaran$ diff copy new_copy -sq
Files copy and new_copy are identical
naran@ace:~/Desktop/CSE-130/adjnaran$
```

- 8) GO TRY BUILDING ONE YOURSELF :)

This project was fun but had a ton of learning curves. We are glad we got working on it early. More to come!