Simple Web Server

Andrei Cristian, andreicristian6@protonmail.com

28-11-2018

1 Introduction

A C-programmed server that can handle multiple clients (web-browsers) at the same time in parallel. The server will send files (.txt or .html with no scripts) from the current directory to any client that connects to it. It will use the HTTP/1.1[1] protocol. [2]

2 Used technologies

The application will use the Transmission Control Protocol (TCP)[3] because what it needs is a reliable process-to-process communication service that can be multiplexed and used for data transfer.
This means we need to be sure that the reciever part of the communication will get all the data that is sent to, but also to be sure that the segments are get in order (reliability is satisfied by TCP by using the ACKs).
The multiplexing is needed because we have a single host (the server) that will serve in the same time multiple clients.

Because a connection might be used for multiple request/response exchanges, the HTTP is used to treat them. I'll include an example[4] of how a typical message exchange for a GET request(Section 4.3.1 of [RFC7231] on the URI "http://www.example.com/hello.txt" will look like:

[1] [RFC7230,7231,7232,7233,7234,7235]
[2] https://profs.info.uaic.ro/~computernetworks/ProiecteNet2018.php
[3] [RFC793]
[4] [RFC7230]

```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
  Host: www.example.com
  Accept-Language: en, mi


Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing CRLF.
```

We can see the server validating the request and sending the response back to the client. I included the above example so you can see how the web server will treat the client request because this type of headers will be used for the server/client communication.

Also, for simplicity, we will only accept requests from User-Agents that are browsers. (ex. User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:63.0) Gecko/20100101 Firefox/63.0)[5]

There could also be fake requests sent to the server(like an application that says it is a browser User-Agent in the request header), but this case will not be treated.[6]

3 Application architecture

The concurence of the server will be implemented by calling the UNIX fork() function in the following manner: when the server *recieves* and *accepts* the client's connection, it *forks* a copy of itself and lets the child handle the client as shown in the figures below:

[5] http://user-agents.my-addr.com

[6] https://makandracards.com/makandra/1613-make-an-http-request-to-a-machine-but-fake-the-hostname

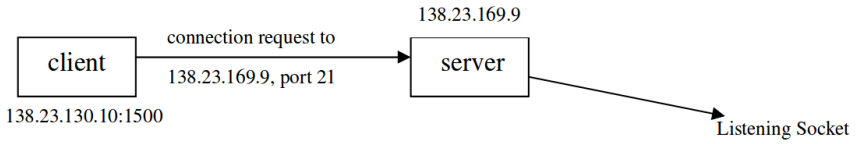

Figure 1: Client with IP address 138.23.130.10 requests to connect to Server (with IP 138.23.169.9 listening from port 21) from its local port number 1500. [7]

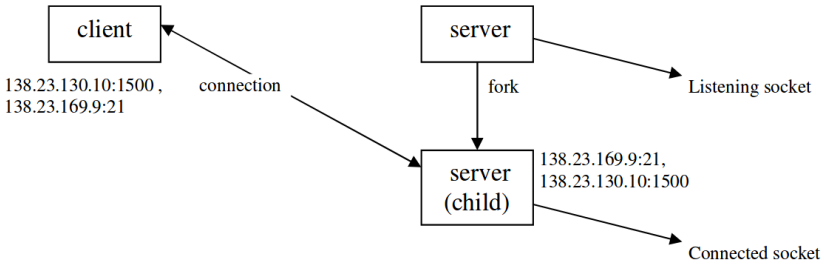

Figure 2: Server fork()s a copy of itself and lets the child handle this client from port 21, it concurrently keeps listening (by the parent process) from port 21 as well.

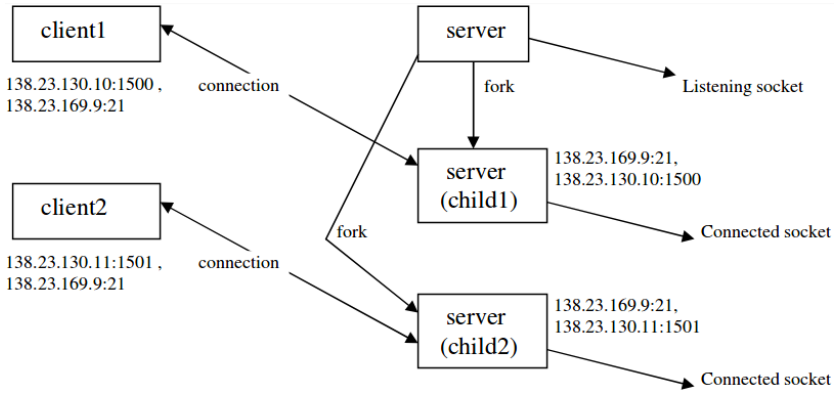

Figure 3: Another client requests to connect to this server. The server fork()s another copy of itself and lets this second child handle this client, whereas it still concurrently keeps listening from port 21 (by the parent process) and keeps serving client1 from port1 (by the first child process).

The **fork** command creates a new separate process for each client called a child. The new process is an almost exact copy of the parent (the process that calls it).[8]

[7] Figures are from cs.ucr.edu/~makho001/masoud/cs164/fork.pdf

[8] cs.ucr.edu/~makho001/masoud/cs164/fork.pdf

For the connection, a **socket**[9] will be used, with the option to reuse the address. This is done before the binding[10]. After the binding step is done, the server will create a listener for the possibly incoming connections[11].

After this step, the application will get the filenames from the directory it is in, plus creating the initial header(the one for the index page), so if anything changes in this directory after the program has started, you'll have to restart the program or you will not see all files/try to access files that will not be there. This means errors.

After this step, the program will accept connections from clients and will serve them by using fork() (hence the concurrency of the server)

All the communication between the server and the client (transfer between files and headers) is done in the child process.

4 Implementation details

Macros used:

```
#define PORT 8080 - port used
#define MAXCONN 10 - max. allowed connections
#define MSGSIZE 90000 - max. file size
#define HDRSIZE 512 - max. header size
#define MAXFNAME 32 - max. file name length
#define NOTFOUND "HTTP/1.1 404 Not Found\r\n"
(for http header creation if it will throw an error)
#define HTTPOK  "HTTP/1.1 200 OK\r\n"
(for http header creation if it will not throw an error)
```

I will explain the functions I've created:

- **char *get_Extension()** - used for getting the file extension from the filename requested; **returns** the extension; used in the *get_ContentType()* function;
- **char *get_ContentType()** - used for creating the HTTP header, based on the extension got with the **get_Extension()** function; **returns**:
  "Content-Type: text/html" if the extension is ".html"/".htm"/".htl"

[9] https://linux.die.net/man/7/socket
[10] https://linux.die.net/man/2/bind
[11] I will use a maximum of 10 connections in this program.

"Content-Type: text/plain" if the extension is ".txt"/".c"/".tex"
"Content-Type: application/pdf" if the extension is ".pdf"
"Content-Type: application/octet-stream" if any other extension is found.

- **int get_FileNr()** - used for getting the number of files in the current directory; useful for creating the *payload*[12]; **returns** the number of files that are in the same directory with the program, exluding itself;

- **char **get_Files()** - used for getting the name of files in the current directory; **returns** an array with the name of the files that are in the same directory with the program, excluding itself;

- **const char *createPayload()** - using it with the values returned from the get_FileNr() and get_Files() as parameters - it creates the file index accesible to the browser (in html format); **returns** payload;

- **const char *createHeader(char *status, int cLength, char* cType,int keepAlive)** - creates the HTTP header;
-status will be one of the created macros: *NOTFOUND* or *HTTPOK*;
-cLength will be the *length* of the *content* (usually the *file size*, or the *size of the payload*, depending for what you create the header);
-cType is the return of the get_ContentType() function;
-keepAlive can be 0 to specify "Connection: close" or 1 to specify "Connection: keep-alive" in the header.It will not be used in this version of the program as the connection will always have the value "keep-alive" for simplicity. **returns** the header;

- **int sendFile(int connection,char *filename)** - send the file with the filename specified by **filename* to the connection to the socket specified by *connection*. If the file exceeds MSGSIZE in size,or it does not exists, or it cannot be opened the function will send a *404 not found header* and will **return 1**; if the file is succesfully sent, it will **return 0**; this function also sends the HTTP headers for the requested file.

- **int verifyRequest(char *buffer)** will **return 0** if the client's HTTP request has "OK" in it, **return 1** if not; this function will not be used in this version of the program for simplicity.

[12]by payload I mean the index page in html format, the page where all the files are listed

5 Conclusions

The HTTP-Request might be forged and can try to access a file that is not listed, meaning access to the system and this always a bad thing. It can be implemented a restriction function that allows only the files in the current folder to be sent. Also, the sending of the file could've been done by sending parts of the file and not the whole file, which is better because it does not use n-bytes of memory (which can lead to worthless usage of the resources)l but less memory at a time. Another bad-thingy would be the implementation of some functions that return a pointer to a local function variable (there have been errors I've stept into from rewriting to that a local function pointer and it gave me headaches but managed to solve them).
Note: I did not check for any memory overflows, or any attacks on the server, but I'm sure it is mostly vulnerable, because the main task was not the security of the server, but to learn how HTTP works.

6 Selected Bibliography

1. HTTP/1.1
    - https://tools.ietf.org/html/rfc7230
    - https://tools.ietf.org/html/rfc7231
    - https://tools.ietf.org/html/rfc7232
    - https://tools.ietf.org/html/rfc7233
    - https://tools.ietf.org/html/rfc7234
    - https://tools.ietf.org/html/rfc7235
2. TCP
    - https://tools.ietf.org/html/rfc793
    - https://profs.info.uaic.ro/~computernetworks/files/4rc_NivelulTransport_En.pdf
3. Fork
    - http://www.cs.ucr.edu/~makho001/masoud/cs164/fork.pdf
    - https://profs.info.uaic.ro/~eonica/rc/lab02.html
    - http://man7.org/linux/man-pages/man2/fork.2.html
4. Socket

- https://profs.info.uaic.ro/~eonica/rc/lab06.html
- https://linux.die.net/man/7/socket
- https://linux.die.net/man/2/bind
- https://linux.die.net/man/2/setsockopt