T-409-TSAM-2017: Computer Networks
Programming Assignment 2 – httpd Part 1

Lecturer: Marcel Kyas

September 14, 2017

Submission Deadline

You must submit the solutions to this assignment through Canvas until 12:00:00 GMT on October 9, 2017.

Intended Learning Outcomes

You should be able to:

- Use the C socket API for TCP connections
- Use mechanisms for event driven communication.
- Understand the HTTP 1.1 protocol

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	1	2	2	2	3	15	10	15	0	50

1 Instructions

Implement a special purpose, in memory HTTP server, a program that generates HTML pages and serves them to a server.

This assignment is Part 1 of a two part assignment. You are required to upload the code and the answers to questions on this sheet. You will extend your solution for Part 2.

Your hand-in must conform to the following to be graded:

- All necessary files must be stored in a compressed archive (zip, zipped tar) that is not larger than 100 KiB.
  - Include a file `./AUTHORS` that includes the name of each group member followed by the e-mail address *at ru.is* enclosed in angle brackets (see example) on separate lines.
  - Include a file `./README` that gives an overview of the structure of your implementation.

  - You must **not** include the data files in `data/`.
  - You must **not** include the file `pa2.pdf`.
  - The unpacked archive should have a directory called `./src` that includes only the necessary source files and a `Makefile`.
  - Do not have those files nested deeper in subdirectories.

- The default rule in `Makefile` shall compile the `httpd` program.
- Do not forget to comment your code and use proper indentation.

We will test your project on a standard RedHat Enterprise Linux Server Version 7.3 (probably `skel.ru.is`). It is a good idea to test your project on such a machine.

2 HTTP Server

A HTTP server serves content using the Hypertext Transfer protocol. The client is usually a web browser. Browsers like Firefox and Chrome include some developer support (e.g., hit F12 in Chrome). Another good debugging aid is `curl` (`https://curl.haxx.se/`) that allows some control over the requests and tracing of responses.

There are very good general purpose HTTP server on the market, like Apache's server and nginx, and also quite a number of special purpose web servers. It is still a good idea to implement custom web servers, as the generic ones do not scale well with dynamic content. This is specifically true for massive multi-player browser based games. The main design concern of these web servers is to minimize processing times by supporting a minimal subset of HTTP.

The HTTP/1.1 was originally defined in RFC 2616. As of June 2014, this RFC is obsolete. In its place RFCs 7230–7235 were released:

- RFC 7230 - Message Syntax and Routing: `http://tools.ietf.org/html/rfc7230`
- RFC 7231 - Semantics and Content: `http://tools.ietf.org/html/rfc7231`
- RFC 7232 - Conditional Requests: `http://tools.ietf.org/html/rfc7232`
- RFC 7233 - Range Requests: `http://tools.ietf.org/html/rfc7233`
- RFC 7234 - Caching: `http://tools.ietf.org/html/rfc7234`
- RFC 7235 - Authentication: `http://tools.ietf.org/html/rfc7235`

HTTP 2 was published as RFC 7540 (`https://tools.ietf.org/html/rfc7540`). We will not consider this protocol, because it is much more complex to implement correctly.

Implementing an HTTP server requires quite some amount of string processing. HTTP messages can be parsed top-down by recursive descent parsers. These have the advantage that they are rather easy to implement.

The C Programming Language does not provide common data structures in its standard library. On Linux, many data types and algorithms have been implemented in GLib 2 (`https://developer.gnome.org/glib/`). Don't use the Fundamentals and the Core Application support functions except necessary (the general memory allocation functions are okay). The utility functions and especially the data types, however, are recommended.

3 Problems and Questions

Your task for this assignment is to program a simple HTTP (Hypertext Transport Protocol) server in C. You are asked to use the C programming language to practice secure programming and understand the security implications of server programming.

For the first part, the focus will be on implementing the methods in a single threaded server. The second part will focus on implementing secure connections, the POST method, and cookie handling.

I suggest to follow the following sequence of steps for implementing the web server.

3.1 Style

It must be possible to run the server using the commands:

```
[student15@skel pa2]$ make -C ./src
[student15@skel pa2]$ ./src/httpd $(/labs/tsam17/my_port)
```

To obtain a port that you can use, call `/labs/tsam17/my_port` on skel. You are free to use any port above 1024 on your own machine.

Make sure that you use a consistent style for your code.

1. (1 point) The source code should be in the directory `./src/`. Use functions to structure the code.
2. (2 points) Take care of a consistent indentation style, meaningful comments, and meaningful variable names.
3. (2 points) Write a Makefile to compile the server.
4. (2 points) Make sure it compiles *without warnings* if using `gcc` using the flags `-O2 -Wall -Wextra -Wformat=2`.
5. (3 points) Make sure that your implementation does not crash, that memory is managed correctly, and does not contain any obvious security issues. Try to keep track of what comes from the clients, which may be malicious, and deallocate what you do not need early. Zero out every buffer before use, or allocate the buffer using `g_new0()`.

   We will use `valgrind` to see if your server allocates memory without deallocating it, or contains any other memory-related errors.

3.2 Single-threaded GET requests

6. Requests from a single client

   Write a sequential HTTP server. HTTP is implemented on top of TCP. The server needs to accept and handle HEAD, GET, and POST requests. It needs to parse the request and reply with a corresponding page.

   For this part of the assignment, your server only needs to handle a requests from a single client at a time. Connections need not be persistent.

   In addition, for each request print a single line to a log file that conforms to the format

```
timestamp : <client ip>:<client port> <request method>
<requested URL> : <response code>
```

where "date" is in ISO 8601 format precise up to seconds, "client IP" and "client port" are the IP and port of the client sending the request for "requested URL" and "response code" is the response sent to the client.

6.1. (5 points) For GET requests, a HTML 5 page has to be generated in memory.[1]. Its actual content should include the URL of the requested page and the IP address and port number of the requesting client, i.e. the visiting browser shall display a line that follows the format

`http://foo.com/page 123.123.123.123:45678`

6.2. (3 points) For a HEAD request, just generate the header of the requested page.

6.3. (5 points) For PUT requests, the page has to be generated in memory and should be a HTML 5 page. Its actual content should include the URL of the requested page, the IP address and port number of the requesting client, and the data in the body of the put request.

6.4. (2 points) For any unsupported request of the server, send an appropriate error.

7. (10 points) Extend your server with support for persistent connections (HTTP keep-alive), i.e., handling several subsequent requests from the same client over a single TCP connection. A persistent connection to a client should be closed after inactivity for 30 seconds. If the client does not ask for a persistent connection if it uses HTTP/1.0 or the client sends the "Connection: close" header, the connection must be closed after sending the response to the client (see Section 6.6 in RFC 7230). After a connection was closed, a new connection from any client should be accepted.

3.3 Parallel connections

8. (15 points) Parallel connections from several clients

   Change your web server to serve multiple clients in parallel. This is best done using the `poll()` system-call, because it imposes the lowest overhead on a machine.

   You must not use `select()`, threads or even processes. Such implementations are incredibly hard to get right, because operating on shared data requires locking. Doing this incorrectly usually leads to *random* errors or random deadlocks. Today, polling incoming messages is combined with polling to generate replies for best performance.

9. (5 points (bonus)) Fairness

   One definition of fairness is that all clients that send a request to the server will eventually receive a reply. Ensure that this is the case for your server and carefully explain why your server ensures this property.

[1] Serving files is a little bit more complex to do securely and will be delayed to Part 2