T-409-TSAM-2016: Computer Networks
Programming Assignment 2 – httpd

Lecturer: Marcel Kyas

September 14, 2016

Submission Deadline

You must submit the solutions to this assignment through until 23:59:59 GMT on October 5, 2016.

Intended Learning Outcomes

You should be able to:

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

Question:	1	2	3	4	5	6	7	Total
Points:	30	10	10	20	30	0	0	100

1 Instructions

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

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

- All necessary files need to be stored in an archive, especially the content of the `src/` directory. If unpacked, we expect to see a `./README` file and a subdirectory `./src/` containing the source code and a `Makefile`; do not have those files in a subdirectory.
- You must include a file called `./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.
- The `./README` file should contain notes about the implementation, like a description of the source code structure, known bugs, and pointers to code that you have doubts about.

- 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 `skel.ru.is`. It is a good idea to test your project on this 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. 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 very 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 As of June 2014, this RFC is obsolete.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`

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

The C Programming Language does not provide a well-rounded standard library. Do not implement your own data types. Instead, use Glib 2.

GLib 2 (`https://developer.gnome.org/glib/`) provides many common data structures, functions for command line argument parsing, functions for more convenient string processing, and so on. 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 Instructions

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.

It must be possible to run the server using this command:

```
[student15@skel pa2]$ make -C ./src
[student15@skel pa2]$ ./src/httpd $(/labs/tsam15/my_port)
```

To obtain a port that you can use, call `/labs/tsam15/my_port`.

Make sure that your implementation does not crash, does not leak memory 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()`.

4 Problems

1. (30 points) 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 GET 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 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
   ```

   For a POST request, a reply page has to be generated that conforms to HTML, like before, but which also includes the data posted.

   For a HEAD request, just generate the header.

   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.

   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.

2. (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 or 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. (10 points) Instead of generating a simple web page, parse the arguments of the request and call functions to generate different content for each request. Parse the query component of the URI (see `https://tools.ietf.org/html/rfc3986`) to supply arguments to the function.

   You do not have to provide a page for every possible URI. Implement one `test` URI that, in addition to the fields displayed above, also displays the queries.

   A second page, called `colour` shall display an empty page, but use the value of the query `bg` as background colour. As an example, requesting the page `http://localhost/colour?bg=red` shall return a page with

   ```
   <body style="background-color:red"></body>
   ```

   as its body.

4. (20 points) Cookies and other headers

   Parse the header of the requests and make the data relevant to users accessible in a useful manner. One method is to provide a dictionary indexed by the header field names. To test this, print out all headers on a web pages.

   Extend the colour web page to set a colour cookie in the client and to remember the colour last requested in the communication. That is, if `http://localhost/colour?bg=red` was requested last, subsequent requests of `http://localhost/color` shall have a red background, until a new color is requested with a new query.

5. (30 points) Parallel connections from several clients

   Change your web server to serve multiple clients in parallel. This is best done using the `select()` system-call, because it imposes the lowest overhead on a machine. You may also use `poll()` to handle multiple connections. On Linux you may use `<sys/epoll.h>`, which seems to be the best performing alternative.

   Do not use 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.

6. (10 points (bonus)) Fairness

   With `select()`, you usually obtain the numbers of file descriptors with pending data. Simple implementations will then search for the lowest file descriptor to serve. This is *unfair* in the sense that under load, connections with low file descriptors are preferred over the ones with higher descriptors.

   Implement a selection mechanism that is *fair*, i.e. whenever a client sends a request, it will not be served a second time before all other requests that were known at the time it was served have been answered.

7. (10 points (bonus)) Speed considerations

   Try to identify where your code spends time. You can use simple timers to identify where your program spends its time. You may use compiler-generated profiling for this task or any other means.

What are the conditions under which optimizing the server implementation for speed will actually result in visible performance gains? How many parallel clients need to be served before the response time decreases noticeably? Is the use of time of your server actually under your control, or is the time spent inside of the operating system?