Computer Networking Graded Assignment n. 1
Università della Svizzera italiana — S. Santini, L. Alecci, M. Laporte March 21, 2022

Assignment 1: HTTP and Web servers

Due date: *Wednesday, April 6, 2022 at 23:59*

This is a group assignment. You may discuss it with other groups, as long as you do not copy their code nor documentation. You must properly reference any and all sources you used. Information about grading and submission guidelines are available throughout and at the end of the assignment (Sections 3 and 4).

Note: extra sessions for help with the assignment. Please, **bring questions**.

- Monday 28.03 - From 17 to 18hrs. Online via Teams. First steps to build a solution. Please read the assignment before this session (and perhaps even try to start a solution).
- Monday 04.04 - From 17 to 18hrs. In-Presence session for quick solving of last-minute problems.

1 Introduction

The objective of this assignment is to develop a Web server. A Web server is a program that *serves* (provides) *Web* resources and satisfies the HTTP requests of clients that want to access those resources. The assignment will cover the expected functionality from the server, as well as some HTTP and Web server concepts that are useful to know.

1.1 Server execution flow

The Web server should behave as follows. A main thread of the server will create a socket and will be infinitely looping, listening for TCP connections. When a new connection arrives, the server accepts it and creates a new thread (this is an *optional task*) that will be in charge of handling the connection.

The new thread will start receiving the HTTP requests through the opened TCP connection. The server must check the correctness of each request and return an error (which one? RFC's to the rescue! ☺) if one of them is **malformed** (missing mandatory headers, unknown methods, etc.). If the request is correct the server must continue and *try* to satisfy it.

If the resource specified in the request does not exist, a new error will be returned (which one?); if the resource exists, the server must respond appropriately with the functionality corresponding to the HTTP method in the request (i.e., the resource will be returned for a GET, only the headers will be returned for HEAD, the resource will be deleted for DELETE, etc.).

If you only do the HTTP/1.0 implementation, the server must close the connection after giving a response; if you also implement HTTP/1.1 (another *optional task*), the connection must remain open waiting either for more requests or for a *Connection: close* header.

The previous explanation can also be modelled with a Finite State Machine (FSM). See Figure 1.

2 Tasks

2.1 Virtual hosts

Although it is easy to imagine sites like Google requiring more than one server to answer all of the world's queries, the converse is also true: sometimes, one server answers requests for more than one site. This is called *virtual*

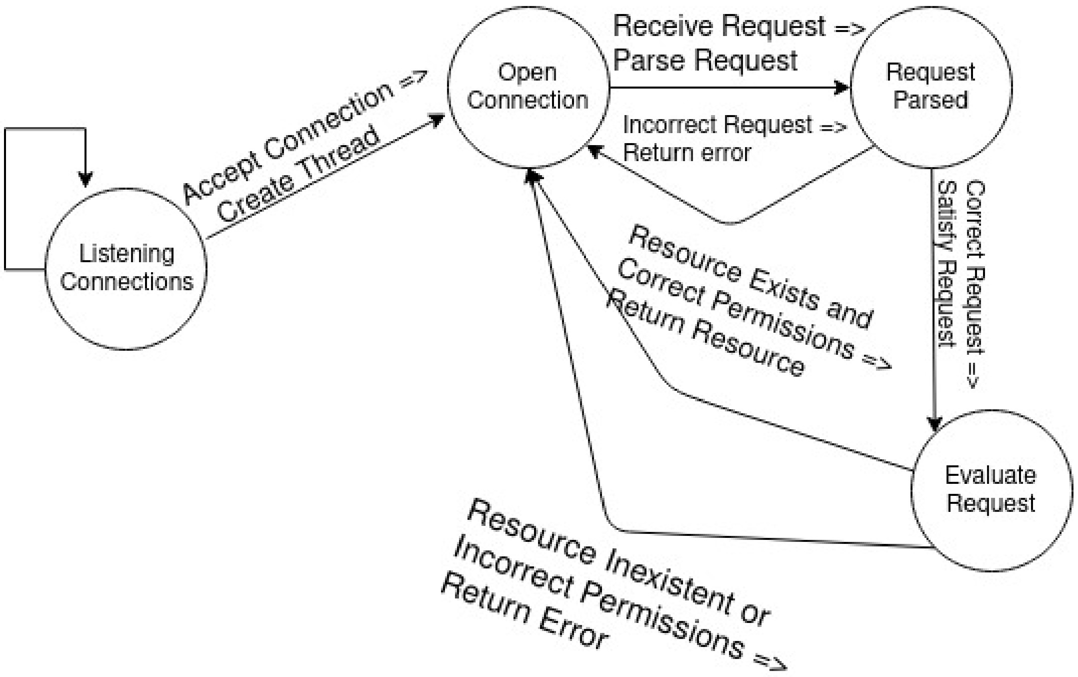

Figure 1: Finite State Machine modeling the server functionality

hosting, and the configuration is usually done through a file. We call this file *vhosts.conf*, and it must be placed in the same folder where the server is executed (as seen in Figure 2).

Your server should host one site per each group member. Each site must be hosted in its own directory, and the name of the directory must match the domain of the site (i.e., if my name were Guy Incognito, and I wanted to access my site at *guyincognito.ch*, the folder with the site's files should have the same name).

Each site must contain at least one HTML file and have an image in its content. The HTML file should include a short text (approximately 400 words) referring to the site owner (the group member), and the image must be related to the text. Creativity is encouraged!

As an example, assume a Web server is created to serve the site *guyincognito.ch*. In the same folder where the server is executed, there must be the *vhosts.conf* file and a folder called *guyincognito.ch*, with the site's files (HTML, picture) inside. Therefore, the file/folder structure (in this example with only **one** site) should be as follows:

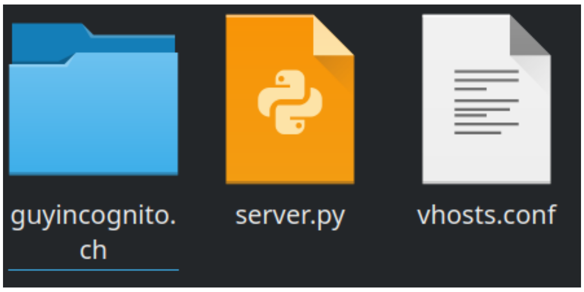

Figure 2: Contents of the server folder

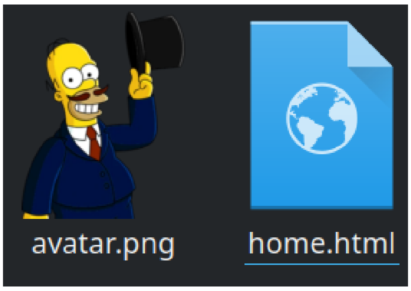

Figure 3: Contents of the *guyincognito.ch* folder

The configuration of the virtual hosts through the *vhosts.conf* must be done in the following way. Each site must occupy one line and be a list of comma-separated values formatted as follows:

domain,entry_point_file,member_fullname,member_email

The entry point of the site in this example is *home.html*. Therefore, if a Web client requests the resource / at host *guyincognito.ch*, the server must reply with the content of the *home.html* file (an identical answer is expected if a client requests */home.html*). The *home.html* file must be placed within the *guyincognito.ch* folder (as seen in Figure 3). Therefore, the line of the *vhosts.conf* file for the example site is:

guyincognito.ch,home.html,Guy Incognito,guy.incognito@usi.ch

Since the assignment asks for three sites to be created (one per group member), the corresponding file should have three such lines. If your group has a different size, you must put that many sites.

Task A (5 points)

The server should be able to read from a *vhosts.conf* file the information regarding the sites it serves.

Task B (10 points)

The submission must include one site per group member.

2.2 HTTP protocols

The Web server should handle at least HTTP/1.0 requests. Optionally, the server can be made capable of handling also HTTP/1.1 requests. If such a functionality is implemented, up to 5 additional points will be assigned for this task (on top of the maximum points available for the task).

Although there are several differences between both versions of the protocol, in the context of this assignment we only care about two:

- HTTP/1.0 expects a new TCP connection for each request; i.e., the TCP connection is automatically closed after receiving an HTTP request and answering it. On the contrary, HTTP/1.1 supports persistent connections; i.e., the connection is kept open and the server receives and answers several requests per connection, until it is explicitly closed.
- HTTP/1.0 has an optional *host* header. For HTTP/1.1, that header is instead mandatory. This means that for HTTP/1.0, if no host is specified, the server must use as default the **first site defined in the vhosts.conf file**.

Optional Task A (10 extra points)

The server should handle HTTP/1.1 connections.

2.3 HTTP methods

The Web server should answer to a subset of the available HTTP methods: GET, PUT and DELETE. Additional information about the functionality of the different HTTP methods is available in the corresponding RFCs for HTTP/1.0 (Section 8 and Appendix D.1 of RFC1945[1]) and HTTP/1.1 (Section 4.3 of RFC7231[2]).

Additionally, the server should also include the NTW22INFO method. Requests for this method will only have / as the resource, and the server should reply like in the following example.

Client (request):

```
NTW22INFO / HTTP/1.0
Host: guyincognito.ch

```

[1] https://tools.ietf.org/html/rfc1945

[2] https://tools.ietf.org/html/rfc7231

Server (response):

```
HTTP/1.0 200 OK
Date: Wed, 21 Mar 2022 09:30:00 GMT
Server: Group ABC Server
Content-Length: 98
Content-Type: text/plain

The administrator of guyincognito.ch is Guy Incognito.
You can contact him at guy.incognito@usi.ch.
```

Note: the data of the administrator of each site must be replaced in the previous message (i.e., **do not** answer with Guy Incognito's name ☺).

Task C (10 points)

The server should be able to answer well-formed GET requests as specified by the corresponding RFCs.

Task D (10 points)

The server should be able to answer well-formed PUT requests as specified by the corresponding RFCs.

Task E (10 points)

The server should be able to answer well-formed DELETE requests as specified by the corresponding RFCs.

Task F (5 points)

The server should be able to answer to NTW22INFO requests as specified by the assignment.

2.4 Request format

This section covers information needed to solve Tasks C-F. Therefore, no specific tasks are defined, but the information given here must be considered for solving those tasks.

Normally the client requests will only consist of 2-4 lines, but the PUT method will also include an entity body. The first line is always the *request-line*, whose format is specified in the RFCs (Section 4 of RFC1945 for HTTP/1.0, or Section 3 of RFC7230[3] for HTTP/1.1): method to be applied, identifier of the resource, and protocol to use. The format was also discussed during Lecture 05 of the class (pay attention to the *blank lines* ☺).

The remaining lines include the headers and the optional body. Although the RFCs define several possible headers for client requests (Section 5 of RFC1945 for HTTP/1.0, or Section 3 of RFC7230 for HTTP/1.1), we are only interested in:

- The *host* header (optional in HTTP/1.0)
- The *Connection: close* header (**only** in the case of trying to end a connection in the HTTP/1.1 protocol, see Section 6.1 of RFC7230). This is related to *Optional Task A*.
- The *Content-type* and *Content-length* headers. When using the PUT method, one should use those headers plus the entity body (which represents the content of the file being put into the server).

[3] https://tools.ietf.org/html/rfc7230

The server should not crash when it finds unexpected headers: it should ignore them (otherwise, you will not be able to test it with a Web browser). If there is no *Content-length* header (meaning there is no message body), the request ends after a CRLF alone in a line (as it can be seen on the 3rd. empty line of the following example). If there is a *Content-length* header, the end of the request is therefore determined after reading that amount of bytes in the message body (after the CRLF).

An example:

```
GET /home.html HTTP/1.0
Host: guyincognito.ch

```

In this case, the server answers with the corresponding HTML file (if it exists). Another example:

```
PUT /new_file.html HTTP/1.0
Host: guyincognito.ch
Content-type: text/html
Content-length: 57

<html>
<body>
Hello! This is a new file.
</body>
</html>
```

In this case, the server creates a new file called *new_file.html* in the *guyincognito.ch* folder, with the specified HTML content. If the file was already existing, then it replaces its content with the new content from the request.

2.5 Response format

This section covers information needed to solve Tasks C-F. Therefore, no specific tasks are defined, but the information given here must be considered for solving those tasks.

The responses from the server should always include the *status-line*. The *message body* at the end should only be answered to a GET request. Regarding *response-header/header fields* (Section 6 of RFC1945, or Section 3 of RFC7231), we only care about the following: *Date* (for GET, DELETE, NTW22INFO), *Content-length* (for GET, NTW22INFO), *Content-type* (for GET, NTW22INFO), *Server* (responding with the name of your group, for all methods) and *Content-Location* (for PUT).

For example, if the client asked for:

```
GET /new_file.html HTTP/1.0
Host: guyincognito.ch
```

The server would answer with:

```
HTTP/1.0 200 OK
Date: Wed, 21 Mar 2022 09:45:53 GMT
Server: Group ABC Server
Content-Length: 57
Content-Type: text/html

<html>
<body>
Hello! This is a new file.
</body>
</html>
```

Note, once again, that there is a CRLF character between the *status-line/response-header* and the *message body*.

2.6 Content types

We will only use the following content types: *text/plain*, *text/html*, *image/jpeg* and *image/png*. You don't have to manage other content types (but obviously you *can*: if you do, let us know in your report).

2.7 Response codes

It is expected that the Web server correctly handles at least (you are free to cover more ☺) the cases that generate the following HTTP status codes (Section 9 of RFC1945, or Section 6 of RFC7231): 200, 201, 400, 403[4], 404, 405, 501, 505.

Task G (10 points)

The server must answer client requests with the correct status code appropriate for the situation. Status codes 200, 201, 400, 403, 404, 405, 501 and 505 must be considered.

2.8 Multithreading

One of the performance metrics used for evaluating Web servers is the number of requests that it can answer per unit of time. A server that can only answer to one request at a time will therefore score quite low in this metric. To obtain good performance (and extra points!) the server should be **multithreaded**, i.e., it should be capable of serving multiple requests simultaneously. You can find a Java example of a multithreaded server available on iCorsi.

Optional Task B (10 extra points)

The server must be multithreaded.

2.9 Extra info

You can always refer to the HTTP/1.0 (RFC1945) AND HTTP/1.1 RFCs (in particular, RFC7230 and RFC7231) for more information about what is *expected* from a Web server and what it *must/should*, *must not/should not*, do.

We also recommend Mozilla Developer Network documents about HTTP[5], where you can find information about the methods[6], the response status codes[7], and the headers[8]. The information is also given in a more user-friendly way than the RFCs (but, of course, may contain mistakes: the RFCs are the ground truth).

2.10 Warning

Make sure to implement the necessary safeguards that prohibit the HTTP server to access or modify filesystem resources that are out of scope from server's folder. This condition is necessary for full points in the DELETE and PUT methods.

[4] Hint: consider OS-level file permissions

[5] https://developer.mozilla.org/en-US/docs/Web/HTTP

[6] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[8] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

3 Submission

3.1 Testing

The Web server should be able to handle well-formed requests from a Web browser[9], but also from command-line tools like netcat, telnet or cURL. Make sure the latter work, as they will be used to test your solution.

In order to test the different sites from your Web browser, where you have no control over the headers sent by the client, you will have to edit the hosts file of your operating system[10], and point the site domain (e.g., *guyincognito.ch*) to the IP 127.0.0.1 (*localhost*). Then you will be able to input your domain in your Web browser and the request should be answered by your server.

3.2 Program Execution

Your program must accept the following command-line arguments:

- –port=*PORT_NUMBER*: although 80 is the well-known port for HTTP requests, it is a restricted (<1024) port that requires sudo. As Instructors do not want to give root privileges to unknown programs, a different default port must be used. Therefore, if no port number is provided as an argument, use port 8080.
- –help: print out a message describing how to use your program.

Task H (5 points)

The server must accept a port number as an argument and it must listen for connections on said port. Use port 8080 as default.

3.3 Available configuration example

You can find on iCorsi, next to this assignment's PDF, a compressed file with the example of a site's content, along with the *vhosts.conf* file. If you put this content in the same folder than your server, it should be able to serve the site. Remember that there must be one site per group member.

3.4 Standard libraries Limitations

You must implement this assignment using the knowledge given in the Lectures, starting from basic socket programming. You may only use **standard libraries**, but some of the modules/packages are forbidden.

None of the following lists is exhaustive, i.e., some libraries could be missing. In case of doubt consult with the instructors, but you should keep the following in mind: we are in a Computer Networking class, therefore if the class/module helps too much with functionality that is pertinent to the assignment (web server, http requests, etc.), then *probably* it shouldn't be used. That's why using a multithreading or socket library is OK, neither of those are the purpose of this assignment (not even of the course).

3.4.1 Standard libraries blacklist

Java:

- HttpServer
- HttpResponse
- HttpRequest
- HttpHeaders

[9] Chrome, Safari, Firefox, Edge

[10] https://www.howtogeek.com/howto/27350/beginner-geek-how-to-edit-your-hosts-file/

Python:

- http.server / http.client (entire module)
- socketserver
- urllib
- SimpleHTTPServer (upgrade your python!)

3.4.2 Libraries whitelist

You can (and should) use libraries to handle the opening/listening/closing of sockets, and the (optional) multi-threading functionality of the server. You can use argparse for the command-line arguments.

3.5 Submission Instructions

You may write your solution in Java or Python (**only version 3**). Add comments to your code to explain the code itself. Package all the source files plus a README file in a single zip or tar archive. **It is explicitly forbidden to include any other files or folders** (e.g. .directory, __MACOSX, thumbs.db, etc.).

The README file is a report of your assignment. Use it to add general comments, to properly acknowledge any and all external sources of information you may have used, including code, suggestions, and comments from other students/groups. If your implementation has limitations and errors you are aware of (and were unable to fix), then list those as well in the file.

The README file must include a list of the group members with the contributions made by each one. Example:

Guy Incognito	Task A, Task B, Optional Task A
Jane Doe	Task B, Task C, Optional Task B
John Doe	Task B, Task D, Task E

Make sure that you include all the necessary documentation and components to build and run your solution on a standard installation of a Java or Python environment (**and without using the HTTPServer classes**). In particular, make sure your solution works with the most basic command-line tools, outside of any integrated development environment.

Submit your solution package through the iCorsi system.

Task I (20 points)

Your code should be in a working state, without the need of modifications from the Instructors, and your submission must comply with the specified guidelines. Points will be deducted if the guidelines are not respected, if the code crashes, has compilation errors or does not execute, or if additional tools are needed for compilation/execution (like maven, gradle, etc.).

Task J (15 points)

Your code must be appropriately documented and your submission must include a README file as a report of your work. The report must include compilation and execution steps, plus a table of contributions.

4 Grading

The assignment has a total of 100 points. If your submission exceeds 100 points, that excess will be added to your future assignments.

Table 1: Score for tasks

Task	Points
A	5
B	10
C	10
D	10
E	10
F	5
G	10
H	5
I	20
J	15

Table 2: Extra score for optional tasks

Task	Points
A	5
B	10

5 Questions and Answers

Q: The RFC states that a reply to a PUT request should return status codes 409 or 415 when there are problems with the request. Should we handle them?

A: Your web server is a subset of what the RFC asks, so we don't expect that you handle those two status codes, but you can if you want.

Q: Can we add the DATE header to a PUT reply?

A: We ask for a minimum. If you want to add things it's OK for us, as long as they respect the RFC. Of course, the RFCs are under-specified in several parts, so they also leave room for the implementers of the standard to decide on things.

Q: If the resource identifier in the PUT method contains a path that doesn't exist, should we create the folder or return an error?

A: There are several distinctions between the concepts of URI/URL/PATH. Theoretically an HTTP Server should not *know* about filesystems, only that a particular identifier (i.e., */new_folder/index.html*) points to a specific resource (the filesystem paradigm is only an implementation choice, you could have a database or something else).

So, if a client does *PUT /new_folder/index.html*, it does not actually matter if you create or not *new_folder*, but if the client does a new request with *GET /new_folder/index.html*, you sould be able to return that resource. Therefore, **probably** (if you are using a filesystem paradigm), it will be easier for you if you just create the folder and put the file there.

Q: What we should do if a request contains duplicate headers?

A: The assignment specifies the type of requests that your server will get, including which headers. None of these headers allow for multiple lines according to the RFC. If the requests don't comply with the RFC, they aren't well-formed requests, so you would need to answer with the corresponding response status code.

Q: Is there an order of priorities for response status codes?

A: The RFC doesn't exactly specify this, so you must decide yourself, as long as you lead the client to solve the problem (i.e., if the client solves one of the errors, then after the next request the answer should reply with the other error).

Q: In tasks C to E it says "well-formed" requests, but we have to reply to the status code 400. How can we do that if the server should only reply to well-formed requests?

A: The assignment doesn't say that you **only** need to answer to well-formed requests. As you are also expected to generate the 400 status code, then you can conclude that you should answer with that code when you get an ill-formed request.

Q: We noticed that there are a lot of edge cases when dealing with requests. The client could send a GET with a body, or a content-length that doesn't match with the body length, or an HTTP/1.1 request without the mandatory Host header

A: the RFCs has answers for most of these kind of questions. Nevertheless, for these cases it is safe to assume that the request doesn't comply with the standard. Therefore, it is not well-formed and you should respond appropriately.