Computer Networks: Project 1

26 February 2015

Submitting your project

Requirements about the delivery of this project:

- Submit via Blackboard (`http://blackboard.ru.nl`);
- Upload one single .zip archive with the structure as described below.

Deadline: Thursday, March 26, 8:00 a.m. sharp!

Goals: On completing this project, you should attain a basic understanding of how an HTTP 1.1 server operates.

Marks: You can score a total of 100 points. The points are distributed as follows:

- 30 points for a basic HTTP 1.1 Server that processes GET requests and returns adequate response;
- 20 points for cache handling;
- 20 points for persistence handling;
- 20 points for test implementation;
- 10 points for documentation.

Implementation: You can use Python 2.7, C, C++ or Java. For C and C++ you need to provide a makefile, for Java you should provide an ant build file. If you use Python, since it is an interpreted language, you don't need any setup file. Make sure that you are using Python 2.7 (and not 3.X or an older version). To check your Python version run: *python −−version*. If it starts with 2.7, it's OK.

We suggest you use versioning. In particular, we recommend Git. A good presentation on Git can be found on Giso Dal's page at: `http://www.cs.ru.nl/~gdal/files/gittutorial.pdf`.

HTTP 1.1 Web Server

Build a Web Server that allows a client/browser, to access resources within a root directory. The Web Server should implement the GET method, whereby content is retrieved from the Web Server. Content can be text or images, which are distributed over a hierarchical structure in your root directory.

More specifically, your Web Server will (i) create a connection socket when contacted by a client (browser); (ii) receive the HTTP request from this connection; (iii) parse the request to determine the specific file being requested; (iv) get the requested file from the servers file system; (v) create an HTTP response message consisting of the requested file preceded by header lines; and (vi) send the response over the TCP connection to the requesting browser. If a browser requests a file that is not present in your server, your server should return a *HTTP 404 Not Found* status (error) message.

Your Web Server should be able to handle concurrent requests. Moreover, since your server supports HTTP 1.1, your Web Server should implement persistent connections. Connections are kept alive unless they time out (with a timeout value of your choice) or are explicitly closed by the client. The timeout value should be signaled in the response via the *Keep-Alive* header. You have to check the *HTTP 1.1 RFC*, `http://tools.ietf.org/html/rfc2616#section-8.1`, for how persistent connections are managed. All the relevant *SHOULD* and *MUST* statements should be implemented. For simplicity, we assume that all clients are HTTP 1.1 compatible.

Your Web Server should also support client side caching. This should be done via the *ETag* header. Using ETags, you build a hash of a resource before retrieving a resource. Then, when the client asks for that resource again, it might send a request with the *If-None-Match* field set to the previous ETag. Your Web Server should rebuild the ETag and check if it is equal to the one received from the browser. In case it is, it should respond with an *HTTP 304 Not Modified* Message. For more information on ETags, check the wikipedia page: `http://en.wikipedia.org/wiki/HTTP_ETag`. As is the case for persistent connections, for ETags you should research the RFC: `http://tools.ietf.org/html/rfc7232#section-2.3`. Your task is simplified, as you can ignore Last-Modified handling when deciding whether to send the content. Obviously, you can still use timestamps when generating your ETag.

In case your Web Server receives a GET for an existing directory, you should redirect to *index.html* in the directory if *index.html* exists (you can use *HTTP 301* response with *Location* set to the new location). In case it doesn't, either build an index of the resources present and return it, or return a *HTTP 404* response. You are free to decide on the course of action, just make sure you document it.

Along with the Web Server, you should also supply a set of tests. The tests should implement the following scenarios:

- GET for an existing single resource
- GET for a single resource that doesn't exist
- GET for an existing single resource followed by a GET for that same resource, with caching utilized on the client/tester side
- GET for a directory with an existing *index.html* file
- GET for a directory with non-existing *index.html* file
- multiple GETs over the same (persistent) connection with the last GET prompting closing the connection, the connection should be closed
- multiple GETs over the same (persistent) connection, followed by a wait during which the connection times out, the connection should be closed
- multiple GETs, some of which are parallel (think of the situation when your browser is fetching a composite resource), the responses should be sent in an orderly fashion

In each case, you should test the response code after every step, as well as the content (if any is expected or if none is expected).

While it is suggested that you use Python, you can, alternatively, use Java, C or C++. In all cases, your server must operate over sockets. You are not allowed to use higher level HTTP frameworks. You will have to provide your own hierarchical structure of resources. For example, a content directory with simple HTML files and a set of images distributed over sub-directories.

The test script runs independent of the server. Implementation of the tests should be done in the same language as the server. For Python, you can use the unittest framework: `https://docs.python.org/2/`

`library/unittest.html`. Note that the test cases resemble common browser behavior in its interaction with a Web Server.

Structure: Your project should have the following structure:

```
proj1_sn1_sn2:
        webserver
        webtests
        content
        run_server.sh
        run_tests.sh
        documentation_file.pdf or documentation_file.txt
```

Where:

- *sn1* and *sn2* are your student numbers (for example, s123456);
- *webserver* is the directory containing your web server code;
- *webtests* is the directory containing your tests;
- *run_server.sh* should build (if needed) and start the server on a given port the script. It should be callable via:

  bash run_server.sh nr_port

- *run_tests.sh* should build (if needed) and run the test scripts on a given port. It should be callable via:

  bash run_tests.sh nr_port

- *documentation_file* should explain the implementation.

In the *documentation_file* (which can be pdf or text), write down how your server is implemented with regards to the protocol specification. That is, you should explain the requirements of the protocol and then briefly explain how you implemented it. Noteworthy requirements: GET request to valid/invalid files, persistent connections and fields relating to that, caching and fields related to that. You should also explain key decisions in your design, namely: the language used, the test framework, the hashing used, setting up the server/tests and others you feel important. Finally, you should mention any difficulties you had in implementing your Web Server.