Computer Networks: Project 1

2 March 2016

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 in the document.

Deadline: Monday, 11 April, 20:00 p.m. sharp! Late submissions can be accepted with a corresponding penalty. Submissions delivered at most one day after the deadline elicit a 15% penalty, submissions delivered at most two days after the deadline elicit a 30% penalty. Submissions are not accepted more than 2 days after the deadline.

Organization: It is strongly encouraged that you work in pairs of two. You can also work individually. Pairs of more than two members are not allowed.

Goals: On completing this project, you should have attained a basic understanding of how an HTTP 1.1 server operates. In particular you should be able to:

- apply your theoretical knowledge of the HTTP protocol in practice;
- create an HTTP 1.1 server that communicates with any standard HTTP requests;
- prepare the web server software for handling common errors.

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

- 25 points for basic GET request processing and response;
- 20 points for persistence handling;
- 15 points for cache handling;
- 10 points for encoding;
- 15 points for tests;
- 10 points for documentation;
- 5 points for interface and structure.

Implementation: You can use the latest Python 2.7 version. A Python framework is provided which is compatible with this version. You are strongly encouraged to build your project on top of this framework. The framework is designed to simplify and streamline construction of the server, and also to give an example of how a test can be implemented.

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`. The faculty's own CnCZ is hosting a GitLab server, accessible with your science login, at `https://gitlab.science.ru.nl/`. For more information on it, see `http://wiki.science.ru.nl/cncz/GitLab`

HTTP 1.1 Web Server

Build an HTTP 1.1 Web Server that gives clients (browsers) access to resources located within a content folder. The Web Server should implement parts of the HTTP 1.1 Specification. More specifically, it should implement (1) concurrent servicing of GET requests; (2) persistent connections whereby GET requests can be serviced over the same connection, pipelining should be implemented; (3) caching using ETags so that content not changed since it was last requested isn't resent; (4) compression by content encoding using *gzip*. All the relevant *SHOULD* and *MUST* statements should be implemented, unless stated otherwise. The implementation should be described briefly in a text document.

The implementation should also be joined by a test suite encoding scenarios covering each of these aspects. The test scenarios are: (1) processing a GET request for an existing single resource; (2) processing a GET request for a single resource that doesn't exist; (3) GET for a directory with an existing *index.html* file; (4) GET for a directory with non-existent *index.html* file; (5) GET for an existing single resource followed by a GET for that same resource, with caching utilized on the client/tester side; (6) multiple GETs over the same (persistent) connection with the last GET prompting closing the connection, the connection should be closed; (7) multiple GETs over the same (persistent) connection, followed by a wait during which the connection times out, the connection should be closed; (8) GET which requests an existing resource using gzip encoding, which is accepted by the server.

For implementing both server and tests you are advised to use the framework joining this document.

Required Functionality

The Web Server should be able to **service GET requests**, by which clients request resources located at a URI. Content can be text or images (jpeg), which are distributed over a hierarchical file structure in your *content* folder. The type of the requested resource should signalled by the *Content-Type* header. In case the requested resource is available, it is returned to the client via a corresponding HTTP response, otherwise a *HTTP 404 Not Found* status (error) message is sent. Processing of GET requests is done *concurrently.* In case your Web Server receives a GET for a directory, you should redirect to *index.html* in the directory if such a file exists. If it doesn't, you should simply return *HTTP 404* response.

Your Web Server should also implement **persistent connections**. Connections are kept alive unless they time out (with a configurable timeout value) or are explicitly closed by the client. Negotiation of a persistent connection is done through the *Connection* header. You should check Section 8.1 of RFC 2616 for how persistent connections are managed. For simplicity, we assume that all clients are HTTP 1.1 compatible.

Your Web Server should support **client-side caching**. This mechanism is guided by the *ETag* header, as specified in Section 2.3 of RFC 2616. Using ETags, you build a hash of the requested resource before sending it to the client. Then, when the same client asks for that resource again, it might send a request with the *If-None-Match* header set to the previous ETag. Your Web Server should then rebuild the ETag of the resource and check if it is equal to the one received from the client. In case it is, it should respond with an *HTTP 304 Not Modified* Message. You can look up HTTP_Etag on wikipedia for more information on ETags.

Finally, your Web Server should support **content encoding** through *gzip*, which is described in Section 3.5 of RFC 2616. Using the *Accept-Encoding* header a client may specify, the acceptable compressions the server should apply on a resource before delivering it. The server should compress the resource by one of these compressions, and in certain cases, signal it using *Content-Encoding* header. If the server does not support any of the accepted compressions, it should send a *HTTP 406 (Not Acceptable)* error response. Your Web Server should support the *gzip* encoding (in addition to the implicit *identity* encoding). You can use wikipedia's HTTP compression example as a reference.

More practical explanations on how to implement this functionality are given in the framework *readme*. The framework already provides assist in implementing some functional aspects, such as concurrency.

Structure and documentation

Your project must adhere to the following structure:

```
proj1_sn1_sn2:
        webserver.py
        webtests.py
        webhttp
        content
        documentation.txt
```

Where:

- *sn1* and *sn2* are your student numbers (for example, s123456);
- *webserver.py* implements the Web Server;
- *webtests.py* implements the test suite;
- *webhttp* is a folder containing all Python resources used by the server/tests;
- *documentation.txt* should explain the implementation.

The *documentation* file should give a brief high level description of how your server is implemented. You should describe in brief the control flow used to implement each functionality by referring to the request/response headers and status codes involved. You should also explain key design decisions, namely: the language used and libraries outside of the framework (if any), how concurrency, hashing and resource encoding are implemented, and other decisions you deem important. Finally, you should touch on any difficulties you had when implementing your Web Server (if any).

Running and options

Your python programs must adhere to the interface below. The framework provided already implements parsing of these options from the command line.

```
# running the Web Server
python webserver.py [-a ADDRESS] [-p PORT] [-t TIMEOUT]
# running the tests
python webtests.py [-p PORT]
```

Where:

- *a* sets the IP address the server is listening to (default is localhost)
- *p* sets the port, default is 8001
- *t* sets the timeout (applied to **all** cases involving a timeout), default is 15

Implementation Details

You are strongly encouraged to use the framework attached with this document, as it will save you plenty of time browsing the Internet for functions/libraries. In any case, your server must adhere to the aforementioned interface.

You **must** use sockets in your interaction with the connecting clients. You **are not allowed** to use HTTP frameworks that implement the internal functionality of an HTTP server. You **are allowed** to use (HTTP) frameworks which help you in parsing the request or composing the response strings. A library that you might find useful is urllib.

Tests are run independently from the server. Implementation of the tests should be done in the same language as the server. Python 2 provides the unittest framework.