King's College London
Department of Informatics

A Secure Web-server

MSc Individual Project - Final Report

Author:
Stefan Peer

Tutor:
Dr. Laurence Tratt

Academic Year 2011/2012

Dedicated to my family

Acknowledgements

I would like to thank everyone who helped me realising this work and supported me during the last year of study.

- My university tutor Dr. Laurence Tratt, who proposed this thesis to me and led me in the right direction.
- My parents, Martha and Arnold Peer, who supported me all the time and allowed me to pursue this study.

Thank you all! Stefan

Abstract

The research project of this thesis comprises the design, implementation and test of a secure web-server. We mainly focused on the implementation of the so called Privilege Separation principles, which state that a program can be split up into several parts with different privilege levels. By applying these principles to a web-server, we wanted to create a system, which behaves in a solid way on attacks of malicious users.
Our server architecture contains three different types of processes, which run with different privileges. In order to obtain maximum security, clients interact just with unprivileged processes, that cannot harm the system. However a privileged process is also needed for performing specific tasks. This process stays in background, isolated from being directly accessed, and cannot therefore be easily taken over by an attacker.
The system was developed using the Python programming language and represents a fully functional web-server, that is able to serve static and dynamic websites. A major challenge during the project has been represented by the asynchronous Interprocess Communication. Privilege Separation split up the program among several processes. Realising the communication between these processes required to take into consideration various issues related to concurrency, efficiency and functionality.
The ultimate goal of this thesis project was to understand, whether Privilege Separation can make a web-server more secure. We compared our system, with the world's most used web-server, Apache. The evaluation showed, that Privilege Separation influences the security of a web-server in a positive way, however affects its performance negatively.

Contents

1 Introduction **1**
1.1 Context of the research 1
1.2 Problems addressed 2
1.3 Approach and Solution 2
1.4 Results of the work 3

2 Review **4**
2.1 HTTP and CGI 4
2.2 Processes and Interprocess Communication (IPC) 6
2.3 Web-server architecture 6
2.4 Privilege Separation 7
2.5 Web-server based on Privilege Separation 9

3 Specification and Design **10**
3.1 Specification 10
3.1.1 HTTP 1.1 11
3.1.2 Virtualhosts 12
3.1.3 CGI and Privilege Separation 12
3.1.4 Output Filters 14
3.1.5 Configuration Files 14
3.1.6 Log Files 18
3.2 Architecture 19
3.2.1 Overview 19
3.2.2 Interprocess Communication 21
3.2.3 Class diagram 24

4 Implementation **27**
4.1 Software and libraries used 27
4.2 Asynchronous requests 27
4.3 Request processing 32
4.3.1 Receiving the request 32
4.3.2 Processing the request 32
4.3.3 Sending the response 35

5 Evaluation **36**

5.1 Research hypothesis 36
5.2 Evaluation strategy 36
5.2.1 Additional aspects 38
5.3 Evaluation results 39
5.3.1 Additional aspects 41

6 Conclusions **44**
6.1 Summary 44
6.2 Future work 45

References **46**

A Appendix: Figures **49**

B Appendix: Configuration Files **55**
B.1 Example of a global server configuration file 55
B.2 Example of a Virtualhost configuration file 57

C Appendix: CGI Scripts **59**
C.1 Site1: tries to deletes all the files from site2 59
C.2 Site2: tries to copy all files from site1 to site2 59

D Appendix: Program Listings **60**
D.1 sws: start, stop, restart script of the server daemon 61
D.2 webserver.py: contains all three types of processes 62
D.3 httprequest.py: HTTP/CGI parser and request handler 66
D.4 config.py: configuration file parser 78
D.5 daemon.py: realises the daemon functionality (open source, public domain) 84
D.6 main.py: PyUnit main class for running all unit tests 86
D.7 servertestcase.py: PyUnit test cases for testing web-server functionalities . 87
D.8 configtestcase.py: PyUnit test cases for testing configuration file parser . . 91

1. Introduction

1.1. Context of the research

The invention of the World Wide Web in 1989 by Sir Tim Berners-Lee, was one of the most important technological occurrences of the last decades. Initially it was developed just for research purposes and only research institutions had access to it [3]. After it was made available to the public, everybody was allowed to add content. It attracted more and more people and a completely new business place emerged. Nowadays we speak about the so called *Web 2.0*, which is a social and business component of peoples everyday life.

The rapid growth of the web was mainly possible due to the fact, that it is an open standard, and therefore everyone was free to contribute, for example by developing web-server software. Today, the Apache HTTP Server, an open source project by the Apache Software Foundation, is the most frequently used web-server worldwide [30]. Since it offers a large set of functionality and is high-performance [16], it will be often used as a reference point in this research project.

While the main purpose of the World Wide Web changed over time, also the requirements for web-server software changed. Initially its purpose was just to publish static resources, i.e., HTML documents, and no complex user interaction was involved. Nowadays the role of the server software is also to provide dynamically generated websites. These are personalised for a specific user, and often allow him to interact with an application in the background (ex. Online Shop, Content Management System, Social Network, etc.). Such systems are usually created using scripting languages like PHP, ASP or JSP.

However, along with this trend, new security risks emerged: hackers and crackers started to put web-applications as well as web-server software into their focus of attention. By exploiting a bug, an attacker could be able to inject malicious code into the web-server software, and might gain illegitimate access to a website or even to the whole system [26]. The consequences of such a break-in could be, in the worst situations, the disclosure of protected data or even the loose of money (ex. Online Banking systems).

The aim of this project is to develop a secure web-server, which behaves in a solid way on attacks of malicious users. Of course, to guarantee that a software is safe, i.e., does not contain faults, is hardly possible, but one can build a software in a way, that in case of a failure it does not open a door to the whole system. This leads us to the concept of the Least Privilege, stated by Jerome Saltzer: every program and every user should operate using the least amount of privilege necessary to complete the job [27].

1.2. Problems addressed

Our research focused on the development of a web-server, which is mainly supposed to fulfil the following two security properties:

1. A website hosted on the web-server should not have access to other websites, hosted on the same server, or administrative access to the whole system - even not in the situation of a failure.

2. A failure in the web-server should not give an adversary the possibility to take over the whole system.

Nowadays there exist many small websites, that are for example blogs or represent companies. These are usually dynamic websites and are often based on a Content Management System. Because of their smallness, with respect to requests per seconds, storage space needed and bandwidth needed, and the performance of nowadays hardware, hosting providers do not need to provide a separate dedicated or virtual server for each single website. One physical server, running a web-server instance, has enough resources for handling dozens of such small websites. It is then the web-server's task to distinguish between the different websites.

However the problem is, that web-servers are usually designed in a way, that they don't separate the privileges of the different websites, hosted on the same server [7]. Consider for example Apache, which is running by default as *www-data* user: every single website that is hosted by Apache, needs to have set its privileges in a way, that the *www-data* user can access it. This means that one dynamic website could easily access content of another website, because both run on the same web-server, under the same user. This reflects a security problem. Apache provides extensions (suEXEC) and configuration mechanisms that try to solve this problem, but they are not enabled by default and are not build into the server's core [10].

The second problem of web-servers like Apache is, that a superuser process is listening for incoming connections. If an adversary would take over this process, because of a bug in the web-server software, he could possibly gain superuser privileges on the whole system [26].

The main task of this project was to come up with a web-server architecture and an implementation that solves these two problems. However, beside the security aspects, we also considered performance to be important for our web-server. The World Wide Web has now hundreds of millions of users [3] and many users visit websites simultaneously. Therefore it is essential for web-servers to handle requests in parallel and as fast as possible, in order treat users equally, and don't let them wait.

1.3. Approach and Solution

Initially we defined the functional requirements for the software. We wanted the web-server to be a cohesive software package, that can also be used in practice. Therefore, all basic web-server functionality, that is needed by nowadays dynamic websites, should be

included. We made a rough sketch of the architecture, in order to plan how to build the server, supporting the security properties, stated in section 1.2. These could be achieved using the so called Privilege Separation principles: different parts of the web-server, i.e., processes, should run with different privilege levels and each part should just gain those privileges it really needs for performing its task [25]. This concept will be explained in more detail in further sections.

Already in the beginnings the Python programming language was selected to implement this project. The main reason for choosing Python was, that it is an object oriented language and therefore allows a clear structuring of the code. Additionally it provides a large set of libraries and system interfaces [19], which are needed for developing server software. It allows programmers to perform powerful tasks without writing a lot of negligible code. Furthermore Python is portable and runs on many operating systems. This project was build and tested on Linux (Ubuntu), however with minor changes it should also run on other systems.

The web-server was developed in an incremental manner: we always tried to have a running system, and introduced new functionality or changed old, step by step. In this way we could compare different solutions, i.e., architectures, and decide which is the best way to proceed. However, because of this approach, a lot of code that was written during development, is not present in the final release. In return the quality of the final software is supposed to be better.

In parallel to the development of the web-server, we created software tests using *PyUnit* [21]. So errors could be detected more easily, without performing regression testing manually.

In the final stage of the project we set up a few sample websites, in order to evaluate the security properties of the web-server. Additionally we made some benchmarks to evaluate its performance.

1.4. Results of the work

At the beginning of this research project, I was not familiar with the Python programming language. I learned it step by step by myself, while developing this web-server. To achieve the desired result, several different programming techniques and concepts have been used, i.e., object oriented programming, process forking, process communication, process privileges, the *select* API, HTTP and CGI parsing, etc. It was all new to me in Python and I sometimes got stuck and had to search new solutions for implementation problems. The course of the project was followed generally as it was planned at the beginning using a Gantt Chart (see Appendix, Figure A.9).

Finally our work resulted in a fully functional web-server, which is based on Privilege Separation principles. Furthermore the software contains all the basic functionality that a web-server needs, in order to deliver dynamic websites, such as support for HTTP, CGI scripts, configuration and log files, error documents, output filters and more.

The evaluation showed, that the server meets the security properties, described in section 1.2 and therefore performs Privilege Separation correctly.

2. Review

In order to facilitate communication over the Internet, a set of communication protocols have been defined by the IETF (Internet Engineering Task Force). This is the Internet Protocol Suite, which consists of four layers, each dealing with different issues, such as routing, reliability, flow control, etc. The four layers are called Link Layer (ex. Ethernet), Internet Layer (ex. IP), Transport Layer (ex. TCP) and Application Layer (ex. HTTP) [5]. A host must implement all these layer in order to communicate using the Internet. In this research project we will mainly deal with the Application Layer, since it is the top layer of the Internet Protocol Suite, and lower layers are handled by operating system or hardware.

2.1. HTTP and CGI

The World Wide Web is an implementation of the Application Layer and is based on a client-server architecture: clients are usually Browsers, which request documents from web-servers. The protocol used is called HTTP and stands for Hyper-Text Transfer Protocol. In this section we want to describe the aspects of HTTP that are most important in the context of this project. The full HTTP specification can be found in the RFC 2616 document [31].

There exist two kind of HTTP messages: requests from client to server and responses from server to client [33].

The format of a request message is a series of newline-delimited lines:

- **Request Line:** request method, URI and HTTP version.
- **Message Headers:** provide information about message transmission and protocols (general headers), about the request (request headers) and about the data (entity headers).
- **Message Body:** data to be transferred (if any).

There are 8 request methods in total, but the three most important ones, that will be considered in this project are:

- **GET:** retrieve resource identified by the URI.
- **POST:** submit resource to the server.
- **HEAD:** identical as GET, but the server must not return a message entity body.

Like the request message, also the format of the response message is a series of newline-delimited lines:

- **Status Line:** HTTP version, status code and reason phrase.
- **Message Headers:** provide information about message transmission and protocols (general headers), about the response (response headers) and about the data (entity headers).
- **Message Body:** data to be transferred (if any).

The status code element is a 3 digit number representing whether the request was fulfilled correctly [34]. The following 5 status code categories exist:

- **1xx Informational:** Request received, continuing process.
- **2xx Success:** The action was successfully received, understood, and accepted.
- **3xx Redirection:** Further action must be taken in order to complete the request.
- **4xx Client Error:** The request contains bad syntax or cannot be fulfilled.
- **5xx Server Error:** The server failed to fulfil an apparently valid request.

Initially, HTTP was just designed for delivering static content, for example HTML files, stored on the server. The URI of a request can be mapped to a location on the server's file system, and the content of the requested file will be returned in the message body of the response. For every request for a specific file, the response's message body will always be the same. However nowadays, as already stated in the previous chapter, websites are mostly dynamic, and they interact with an application in the background. They are personalised for individual users (ex. shopping cart of an online shop), and the content therefore cannot be simply retrieved from a file, but must be generated dynamically by a script or an application (ex. PHP, ASP, JSP, etc.). A standardised protocol for achieving this dynamic is CGI, which stands for Common Gateway Interface. Its current version 1.1 is defined in the RFC 3875 document, and basically describes how a web-server can delegate a HTTP request to an executable file, which then generates the response [17].

The server is responsible for managing connection, data transfer, transport and network issues related to the client request, whereas the CGI script handles the application issues, such as data access and document processing [17]. On a request, the server determines the script to be executed, by the request URI, and prepares a set of so called meta-variables like, *SERVER_NAME*, *REMOTE_ADDR*, *REQUEST_URI*, etc. It then invokes the script as an executable program, and passes all meta-variables as system environment variables to the script. The optional message body of the HTTP request will eventually be provided to the script at the standard input file descriptor.

The CGI response can have several formats, but the most important one, to be mentioned here, is the document response: the CGI script provides the content, that is supposed to be delivered back to the client, at the standard output file descriptor [17]. The web-server fetches it from there, and packs it into a HTTP response message.

2.2. Processes and Interprocess Communication (IPC)

A process is a program instance, that is currently being executed. Every process is running with the privileges of a particular system user and has its own memory address space, which can not be accessed by other processes [29]. On Unix systems, processes are created using the *fork* system call [24]. It creates an exact copy of the calling process, so that both, the parent and the new child process, afterwards, have the same memory image and the same open file descriptors. However, their address space is different. So tasks, that will be executed after the *fork* on one process, do not affect the other process.

There exist processes called daemons, that stay in background, detached from any terminal, and provide services, such as web-servers, mail-servers, etc.

Software is often build out of several processes, for achieving parallelism and therefore better performance, but also for security reasons (see section 2.4). In this case, different processes often need to communicate with each other, but since they have separate address spaces, direct memory accesses between them are not possible. Therefore, modern operating systems provide several methods for Interprocess Communication (IPC) [28], such as:

- **Shared Memory:** A memory space in RAM, that is shared between processes, i.e., all processes can read and write.
- **File:** A simple file stored on the file system. Data can be read and written to the file by all processes that have access to it.
- **Pipe:** A FIFO data stream between two processes. If the stream is bidirectional, both processes can send and receive data.
- **Socket:** Similar to a pipe, but supports also communication over a network, between processes that are on different hosts.
- **Signal:** Signals can be sent to processes, in order to notify them, that an event has occurred.
- **Remote Procedure Calls:** Middleware that deals with communication issues between processes.

2.3. Web-server architecture

In this section we want to give a brief insight in the architecture of a web-server, referring generally to the Apache 2 implementation. As already described in earlier sections, web-servers communicate using the HTTP Application Layer protocol. This sits on top of the TCP/IP protocol, therefore a web-server is also a TCP server, and listens on a given port (ex. 80) for incoming connections. Clients, i.e., Browsers, establish a TCP connection to that port, for being able to issue their HTTP request and getting back a HTTP response.

Considering Apache 2, connection management and request processing is not all done by a single process, but by a multiprocess architecture. This has several advantages, such as higher performance, better stability and better availability [29, 37]. The Apache

web-server comes with different multiprocessing modules (MPMs), which are mainly responsible for the binding to network ports, accepting requests and dispatching children to handle the requests [14]. For our research the *Prefork* module was the most relevant, which is also the default on Unix. It is based on the fact, that using the *fork* system call, it is possible to create identical copies of existing processes, i.e., a parent process spawns child processes.

In Apache's architecture, there is one parent process, that is a superuser process, and listens for incoming connections. This process is often also called listener or control process. It is mainly responsible for launching child processes and delegating requests to them. All the request processing is done by these child processes, which minimises the work load for the listener. This approach improves concurrency, since the listener, after request delegation, is immediately ready for accepting new connections. Another performance improvement is realised by the pre-fork mechanism: the control process maintains several spare child processes, which stand ready for serving incoming requests immediately [15]. This helps to reduce the delay time, the operating system needs to execute the *fork* system call [24].

The request processing stage, that is done by the child processes, can be very complex, depending on the modules that are used. There are several phases, between the arrival of a request and the delivery of a response. The main phase is called content generation [6], during which generally either a static file will be retrieved or a CGI script will be executed. Before that phase, the request is passed through several other stages, such as parsing the request headers, finding a matching Virtualhost, checking whether the requested file is jailed into the document root directory, checking the existence of the file, etc. After the content generation phase there can be also several stages, such as logging of the request or modification of the output data using filters.

A very important aspect for this project is related to the content generation phase, and refers to the access privileges of files on the file system. All the child processes, that handle requests, run under the same system user (ex. *www-data*, see Figure 2.1). This user must be able to have access to all files served by the web-server [15]. This exigence introduces a security problem, because the privileges for all websites, running on the same web-server, are therefore the same, and unrestricted file accesses across websites are possible, via CGI scripts.

2.4. Privilege Separation

As already explained in previous sections, services that are accessible from the Internet, are generally vulnerable to attacks from malicious users. They often try to exploit a bug, which gives them control over the system. There are several approaches that help to find or avoid bugs, such as software testing or the use of type-safe programming languages [36]. However, it is hardly possible to guarantee that a software contains no errors. Even with excessive software testing, there might still remain an undetected bug, and therefore nobody knows what could possibly happen if it will be exploited. However, one can try to minimise the consequences of a software failure, by following

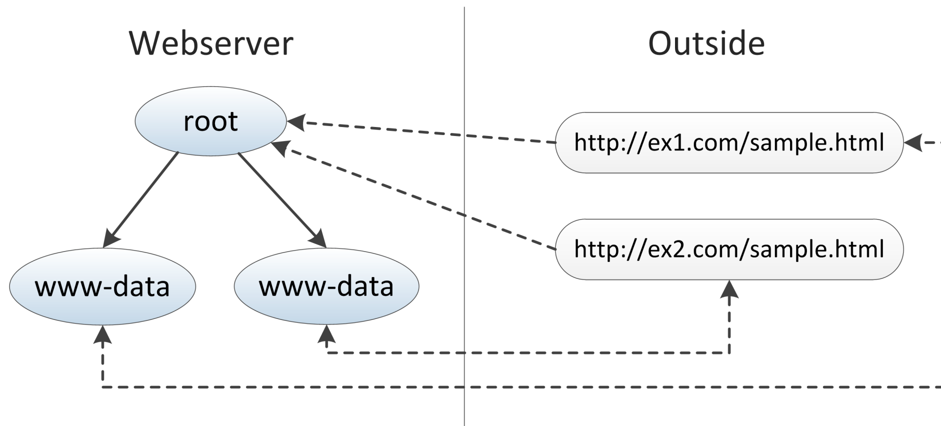


Figure 2.1.: Request processing of Apache: a superuser (root) process is listening for incoming connections, and forks request handling processes, that run under the *www-data* user.

the concept of the Least Privilege, stated by Jerome Saltzer: every program and every user should operate using the least amount of privilege necessary to complete the job [27]. A program that runs with no privileges will harm, even in case of a failure, the system less, than a program with superuser privileges. However, simply running a program with no privileges might not always be possible, since some tasks of a program often need superuser privileges. This leads us to the concept of Privilege Separation.

Privilege Separation describes a programming style, where a program is split up into several parts which have different privilege levels. One can distinguish between privileged parts, called monitors, and unprivileged parts, called slaves [2]. Usually an application is build of several slaves, but just one monitor. Slaves can perform just operations that do not require privileges, while for the monitor these restrictions do not count; usually the monitor has superuser privileges. However, if a slave needs to perform an operation that requires privileges, it has to ask the monitor to do this. Communication between the monitor and the slave happens via a well defined channel, and the monitor, prior to execute the requested operation, validates it and checks if it is permitted. After the execution the result is communicated back to the slave [25].

This architecture usually increases the complexity of software, but it does not limit its functionality. The advantage is, that the amount of code, that is executed with superuser privileges, can be reduced [25]. This in turn limits the scope of software errors: if an adversary takes over an unprivileged slave, because of a programming bug, there is no way to execute illegitimate code, that requires privileges. So the adversary controls just an unprivileged part of the application and cannot break out of this part, for taking over the entire system. However, if an adversary would take over the monitor, this would introduce a security problem and might lead to the illegitimate execution of privileged code. Therefore, the monitor should be isolated from the outside, i.e., not accessible directly from the Internet.

2.5. Web-server based on Privilege Separation

The main goal of this project was to solve the problems of existing web-servers, like Apache, that have been described so far, by building a secure web-server, which is based on Privileges Separation. In this context, privileges can be separated at process level, since each process runs as a system user, and therefore has predefined privileges [29]. This changes a few aspects in the web-server architecture. Processes which are vulnerable for being taken over, i.e., do interact directly with users or clients, should be unprivileged and have just those privileges that they really need [27]. In this way, in case of being taken over, they cannot harm the system as much as privileged processes could. We stated in section 2.3, that Apache, in combination with the *Prefork* module [15], is build out of two kinds of processes: unprivileged request handling processes and a privileged listener/control process.

Applying Privilege Separation and the concept of the Least Privilege to a web-server means, that a request handling process should detect the privileges of the website it serves, and run with privileges, that grant access to just this website. This stands in contrast to Apache's architecture, where all request handling processes run with the same privileges (ex. *www-data*). So, for example, if a process' responsibility is just to serve website A, it should not have access to website B.

Additionally, the listener process should be unprivileged, since it is accessible directly via the Internet. However, in this case, all parts of the web-server would be unprivileged. This leads to the problem, that tasks, that require privileges, such as process forking or the binding to network ports, are not executable any more. Since those are essential for a web-server, we had to come up with an architecture that solves this problem.

3. Specification and Design

3.1. Specification

The main goal of this project was the development of a secure web-server, that is based on the Privilege Separation principles. These have already been explained so far. We just wanted to point out here the two main points:

1. A website, i.e., a CGI script, hosted on the web-server should not have access to other websites of the same server, or administrative access to the whole system, even not in the situation of a failure.

2. A failure in the web-server should not give an adversary the possibility to take over the whole system. Therefore the process connected to the Internet, i.e., listening for incoming connections, should not have superuser privileges.

However in addition to these security aspects, the web-server was supposed to be fully functional, so that it can be used in practice and serve nowadays dynamic websites. We took the Apache server as an example and selected a small subset of its functions, that we considered as relevant for this project.

The most basic requirement is the support for the HTTP 1.1 protocol, specified in the RFC 2616 document [31]. Beside the delivery of static content, the server was considered also to support CGI scripts (RFC 3875 [17]), so that both dynamic and static websites can be served. In order to test the security features, being explored in this project, the server should also support Virtualhosts, i.e., support the hosting of several different websites.

Additionally we wanted the web-server to be a cohesive software program, which is able to run as a system service (daemon). It should fetch, at start up, global and website specific settings from configuration files, and should log website accesses and errors, in real time, in separate files. The web-server should also provide error documents, using HTTP status codes (4xx, 5xx), if a resource could not be retrieved successfully. It is also supposed to support Output Filters, i.e., scripts, that are able to modify the HTTP response body, before it gets returned to the client. All these requirements will be further explained in the next sections.

Since we wanted the web-server to be used in practice, we also considered performance as important. However, this was not a main requirement for this project, because we mainly focused on security.

3.1.1. HTTP 1.1

In order to meet the HTTP standard, set by the Internet Society, the web-server to be developed has to comply strictly with the the HTTP 1.1 specification, defined in the RFC 2616 document [31]. However, because of the extensiveness of this specification and the relative short project duration, only the most relevant parts of it have been considered for being implemented.

Request

The RFC document specifies 8 request methods, such as *GET*, *POST*, *PUT*, etc. However, nowadays mainly three of them are used for retrieving and submitting resources, and therefore only those have been considered as relevant for this project: *GET*, *POST* and *HEAD*. The purpose of each is explained in section 2.1 or in RFC 2616.

The web-server is supposed to accept any HTTP request header. Headers are optional, except the *Host* header, which is obligatory in some cases and specifies the internet host (ex. sws.peerweb.it), and the port number (ex. 80) of the resource being requested [32]. If available, HTTP headers such as *User-Agent* or *Referer* should be used for logging purposes.
The following shows an example request:

GET /index.html HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:14.0) Gecko/20100101 Firefox/14.0.1

Response

The HTTP specification defines 5 status code categories, and each category specifies several status codes. In this project, the following codes have been considered as relevant:

- **200 OK:** the request was successful.
- **301 Moved Permanently:** the client should forward automatically to the URL specified in the *Location* response header.
- **400 Bad Request:** there was an error in parsing the request, because of a wrong syntax or a missing header.
- **403 Forbidden:** the requested resource was found, but not allowed to be accessed.
- **404 Not Found:** the requested resource was not found.
- **500 Internal Server Error:** a server error occurred, or there was a problem with a CGI script.

In case of a 403, 404 or 500 error, the server should return user defined error documents, i.e., a static or dynamic web-page, instead of a predefined error message in plain text. Error documents can be specified in global or website specific configuration files. More about that will be explained in section 3.1.5

Like for the request, also the HTTP response includes several headers. The server is supposed to set the following headers automatically:

- **Connection: close:** HTTP 1.1 supports persistent connections, however, this feature is not included in this project. Therefore, the web-server supports just one HTTP request per TCP connection, and closes the connection after each request.
- **Server: SWS/1.0:** the name and the version of the server (SecureWebServer/1.0).
- **Date:** the current date and time of the server.
- **Content-Type:** the content type of the data in the response body.
- **Content-Length:** the length of the content in bytes (optional).
- **Location:** in combination with the 301 status code, defines the location for a client redirect.

However consider, that CGI scripts can influence the HTTP response. Therefore it is also possible that other status codes or headers occur. Additionally it is possible to specify response headers in configuration files, that will be set automatically in any HTTP response, that matches a particular context (see section 3.1.5).

3.1.2. Virtualhosts

The web-server is supposed to be able to handle more than one website. Consider for example two domains, *site1.sws.peerweb.it* and *site2.sws.peerweb.it*, pointing both to the same server, but containing different websites. In order to serve these websites, the server must first be able to distinguish the two host names, and then be able to know where to find the files of each website, i.e., locate its document root directory on the servers file system.

The host name can be identified either by the *Host* header field of the HTTP request (ex. *Host: site1.sws.peerweb.it*), or by the request URI, if it matches the following format: *http://site1.sws.peerweb.it/page.html* [35].

In order to assign a host name to a document root directory on the server, so called Virtualhosts can be specified in configuration files. They assign one or more domain names to a directory on the server, inside which all files and subdirectories are supposed to be stored [11]. Consider, that for security reasons, the web-server has to make sure, that a user can't access files outside of this directory.

3.1.3. CGI and Privilege Separation

In order to be able to generate web pages dynamically, just at the moment of the request, the web-server is supposed to support CGI scripts. How this works has already been explained in section 2.1. The web-server is supposed to meet fully the CGI 1.1 specification, defined in the RFC 3875 document [17], therefore we do not describe details of this here again.

On every HTTP request, the server must be able to distinguish whether the requested file is a CGI script or a static resource, since they have to be treated differently. One

possibility would be to check, if the resource is an executable file, and handle it as CGI script in that case. But this might not be a good solution in any case, because the executable file could also be an application, that is offered for download, or be executable by mistake. A better solution is to define contexts, in which files are handled as CGI scripts. This can be done by introducing Virtualhost-specific configuration directives, like a *Directory* of the Apache implementation [8]. In this way, one can, for example, define the following context: a directory, in which all files with a specific extension are handled as CGI scripts. Files, that do not match this context, are handled as regular documents.

Furthermore, CGI scripts need some special treatment by the server, because they could contain errors, or could try to perform illegitimate actions. A script should not affect the behaviour and the stability of the web-server or other websites. As already described earlier, Privilege Separation is considered the main solution to this: different websites are served by processes with different privileges. Provided that these privileges are defined properly, no interference is possible, because processes are subject to restrictions of the operating system.

There are several approaches to include Privilege Separation into a web-server. The basic idea is to have for each Virtualhost a different user, that is going to serve the website. Therefore, all files and folders in the document root directory must have set the privileges properly, so that just the serving user can access them. In order to inform the web-server about the privileges of each Virtualhost, we came up with two different ideas. The first was to specify for each Virtualhost the user-id and the group-id in the configuration file. However, we realised soon, that this is not necessary and just makes the system inflexible. One would have to apply first the privileges to all files and folders, and then specify them again in the Virtualhost settings. This leads to redundancy and makes the system more complicated to configure, and more vulnerable to misconfiguration. So we agreed, that it is more simple, if the web-server detects the owner and the group of the requested file, and sets the privileges of the handling process accordingly.

Regarding CGI scripts, there are also error related issues, that have to be considered. What if a developer introduces an endless loop in a CGI script? In this case, the script process would consume system resources, that won't be released until the server gets restarted. This problem is supposed to be solved the same way as in Apache: CGI scripts get killed by the server after a specific time-out, which can be defined in configuration files. Note, that for slow scripts, one should set the time-out value high enough. If an aborted or failed CGI script hasn't streamed yet any response data to the client, the server should respond with an Internal Server Error (500). In case some data has already been sent out, the web-server should just close the connection.

Further issues with CGI scripts are related to file properties. What if a CGI script is not an executable file? What if it is owned by the superuser? A CGI script which is not executable, should just return an Internal Server Error (500), like Apache does it.

As already explained earlier, the web-server is supposed to detect the owner of a requested file, for determining the privileges of the handling process. Consider, that CGI

scripts should never be executed with superuser privileges, because this would introduce security problems. If a CGI script is owned by the superuser, one option would be to return an Internal Server Error (500) and forbid the execution. However, we considered this as a too strict constraint, and decided to run those CGI scripts under a predefined user, that can be specified in a configuration file.

3.1.4. Output Filters

The resulting web-server is also supposed to support so called Output Filters [9]. These are scripts, that can be chained one after the other (pipe-lining), and the HTTP response's body data is passed, i.e., filtered, through them, before finally returned to the client. Every filter script gets the input data on *stdin* and is required to write the output data on *stdout*.

Output Filters allow, for example, compression (gzip) or other data modifications, right before the HTTP response will be sent. In most cases the presence of a filter is not apparent to the client. Filters are supposed to be defined in configuration files.

Consider that the Privilege Separation principles do also apply for Output Filters: filter scripts have to be executed by the owner of the requested resource, not to confuse with the owner of the filter script.

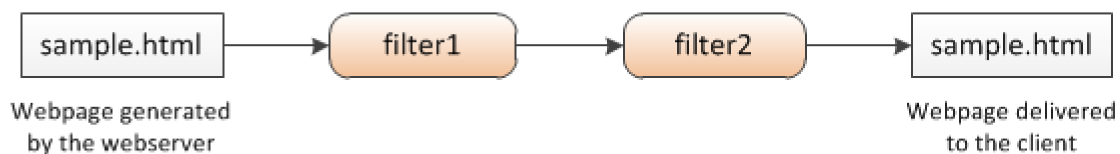


Figure 3.1.: Example of a configuration with two Output Filters

3.1.5. Configuration Files

The web-server, to be developed in this project, can be configured using external configuration files. There is one global configuration file for the whole web-server, and separate configuration files for each Virtualhost. The structure of these files is very similar to Apache's.

Global web-server configuration directives

- **Listen:** Port on which the server is listening for incoming connections.
  Default value is 80.
- **Host:** Host on which the server is listening for incoming connections.
  Default value is 0.0.0.0.
- **User:** User name or user id of the listener process. This user will also be used for request handling child processes, if the file owner of the requested resource cannot be determined or is the superuser.

- **Group:** Group name or group id of the listener process. This group will also be used for request handling child processes, if the file owner of the requested resource cannot be determined or is the superuser group.
- **HostnameLookups:** Determines whether the server performs hostname lookups, i.e., maps IP addresses (ex. 46.4.17.148) to hostnames (ex. sws.peerweb.it). This should be turned off for performance reasons.
  Default value is Off.
- **DefaultType:** Content type, that the server delivers for documents, whose type could not be determined.
- **AddType:** Allows to manually add *file extension* to *content-type* associations. It causes the server to set for all files with the specified extension, the specified content type.
  This configuration directive can be used multiple times.
  Usage: AddType .css text/css
- **CGITimeout:** Seconds after which CGI scripts get aborted automatically, if not terminated yet.
  Default value is 30.
- **CGIRecursionLimit:** Specifies for a client request, the maximum number of allowed, sequential, server redirects in CGI scripts. This is to prevent endless loops in redirections.
  Default value is 10.
- **ErrorDocumentRoot:** Directory which contains all the error documents.
- **ErrorDocument:** Specifies for a given status code, the error document, that the server will provide in case of such error.
  This configuration directive can be used multiple times.
  Usage: ErrorDocument 404 notfound.html
- **ErrorLogFile:** Path to the standard error log file.
- **AccessLogFile:** Path to the standard access log file.
- **CommunicationSocketFile:** Path to the file, which will be used as Unix socket for process communication.
- **ListenQueueSize:** Size of the listen queue for incoming connections, i.e., number of requests that can be accepted simultaneously by the listener process.
  Default value is 10.
- **SocketBufferSize:** Size of the communication buffer, used for Interprocess Communication, in bytes.
  Default value is 8192.

An example configuration file is shown in the Appendix B.1.

Virtualhost configuration

Every Virtualhost configuration must be stored in a separate configuration file. Each file must at least contain the following two directives:

- **DocumentRoot:** Root directory for the documents being served by this Virtualhost.
- **ServerName:** Main hostname which maps to this VirtualHost.

Exactly one Virtualhost configuration file must contain the **DefaultVirtualHost** directive, which specifies the default Virtualhost. It will be used by the web-server if no other matching can be found, i.e., the host, specified in the HTTP request, was not found on the server.
Further configuration directives are:

- **ServerAlias:** Additional hostname, beside the main *ServerName*, which also maps to this Virtualhost.
  This configuration directive can be used multiple times.
- **ServerAdmin:** E-mail address of the person, that is responsible for this Virtualhost.
- **ErrorLogFile:** Path to the error log file for this Virtualhost.
  If not specified, the error log file, specified in the global configuration file, will be used.
- **AccessLogFile:** Path to the access log file for this Virtualhost.
  If not specified, the access log file, specified in the global configuration file, will be used.
- **ErrorDocumentRoot:** Directory which contains all the error documents for this Virtualhost.
  If not specified, the *ErrorDocumentRoot*, defined in the global configuration file, will be used.
- **ErrorDocument:** Specifies, in the context of the Virtualhost, for a given status code, the error document, that the server will provide in case of such error.
  This configuration directive can be used multiple times.
  If not specified, for a given error code, the corresponding error document, defined in the global configuration file, will be used.
  Usage: ErrorDocument 404 notfound.html
- **ExtFilterDefine:** Defines an output filter script: assigns a name (ex. filter1) to a script (ex. /bin/filter.sh).
  This configuration directive can be used multiple times.
  Usage: ExtFilterDefine filter1 cmd="/bin/filter.sh param1"

Configuration directives sometimes should be valid just in a particular context, and not for the whole Virtualhost. Because of the limited time available for this project we considered just one type of context, which is *directory.* A *directory* can be defined using a tag: <*directory "/dir/subdir"*>. All directives, listed between the opening and the closing tags, are just valid for the specified directory and all subdirectories. The following configuration directives can be used inside a *directory*:

- **DirectoryIndex:** Defines a list of files to look for, if the requested resource is a directory.
  This configuration directive can be used multiple times.
  Usage: DirectoryIndex index.php index.html index.htm
- **CGIHandler:** Defines a filename extension that will be considered as a CGI script. Optionally also the executing program for the script can be defined (ex. bash, php-cgi, etc.).
  This configuration directive can be used multiple times.
  Usage: CGIHandler .php /usr/bin/php-cgi
- **SetOutputFilter:** Defines a chain of output filters, that are applied to every request that matches this context.
  Usage: SetOutputFilter filter1;filter2;filter3
  Consider that filter1, filter2 and filter3 must have been defined previously using *ExtFilterDefine.*
- **AddHeader:** Allows to define a response header, that will be automatically set in every response.
  This configuration directive can be used multiple times.
  Usage: AddHeader "Content-Encoding" "gzip"
- **AddType:** Allows to manually add *file extension* to *content-type* associations. It causes the server to set for all files with the specified extension, the specified content type.
  This configuration directive can be used multiple times.
  Usage: AddType .css text/css

There is one further directive, that can just be used inside a *directory* tag, which is called **StopInheritance**. It can be defined for a subdirectory, and allows to stop the automatic inheritance of a directive. Consider for example a /filter directory, for which an Output Filter was defined. The filter would also be valid for a /filter/pictures directory, since it is a subdirectory. However, defining "*StopInheritance SetOutputFilter*" inside the /filter/pictures *directory*, removes the output filter setting for this directory and all its subdirectories.

StopInheritance can be used multiple times, with parameters such as "All", "DirectoryIndex", "CGIHandler","SetOutputFilter", "AddHeader" and "AddType".

An example configuration file for a Virtualhost is shown in the Appendix B.2.

3.1.6. Log Files

Since the server is running as a daemon, eventual errors cannot be shown on standard output, but have to be logged. Two kinds of log files should be supported: access log and error log. The access log file stores information about every document request. The error log file contains CGI script errors, file not found errors, forbidden errors, etc. The structure of these files is exactly the same as Apache's [13], which means that log analyser tools that work for Apache, can also be used for this server. An example for this is Webalizer [1] (see Appendix, Figure A.8).

Access Log File

The access log file is structured in the following way:
hostname:*serverport* *client IP* - - [*request time*] "*request URI*" "*referer*" "*user agent*"

```
root@ubuntu:/home/stefan/sws/log# tail access.log
localhost:80 127.0.0.1 - - [20/Jul/2012:10:14:53 +0100] "GET /pic.jpg HTTP/1.1" 200 59213 "http:
//localhost/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/12.0"
localhost:80 127.0.0.1 - - [20/Jul/2012:10:14:54 +0100] "GET /cgi-bin HTTP/1.1" 200 0 "http://lo
calhost/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/12.0"
localhost:80 127.0.0.1 - - [20/Jul/2012:10:14:57 +0100] "GET / HTTP/1.1" 200 213 "-" "Mozilla/5.
0 (X11; Ubuntu; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/12.0"
```

Figure 3.2.: Three example log entries of an access log file

Error Log File

The error log file is structured in the following way:
[*request time*] [*error level*] [client *client IP*] *error message*

```
root@ubuntu:/home/stefan/sws/log# tail error.log
[Fri Jul 20 03:56:28 2012] [error] [client 127.0.0.1] cp: cannot create regular file `/home/stef
an/sws/docs/site1/2012-07-20-03-56-28.html': Permission denied
[Fri Jul 20 03:56:35 2012] [error] [client 127.0.0.1] 404 Not Found: /home/stefan/sws/docs/site1
/2012-07-20-03-56-28.html
[Fri Jul 20 04:01:51 2012] [error] [client 127.0.0.1] 403 Forbidden: /home/stefan/sws/docs/site1
```

Figure 3.3.: Three example log entries of an error log file

3.2. Architecture

3.2.1. Overview

In this section we want to show how we designed a web-server, based on Privilege Separation and the concept of the Least Privilege. In section 2.3 we described the basic structure of Apache and which processes are involved in serving requests. There is a privileged listener process, which is responsible for accepting new connections on the server-port (usually 80). It delegates incoming requests to a pre-forked, unprivileged, request handling process, which generates the HTTP response (see Figure 2.1).

However, this architecture does not meet our definition of a secure web-server because:

1. A website, i.e., a CGI script, hosted on the web-server could access other websites, hosted on the same server.

2. A superuser process represents the interface to the Internet: a failure in this process could possibly give an adversary administrative access to the whole system.

In order to comply with our definition, we modified some concepts of Apache's architecture. First of all, an unprivileged slave process is introduced, which is connected to the outside and listens for incoming connections, instead of a superuser process. However, a superuser control process is still needed for forking child processes that handle requests. This so called monitor process stays in background and communicates with the slave process via a defined communication channel (see section 3.2.2). Every valid request will be forwarded by the slave to the monitor, which then delegates it to a child process for handling it.

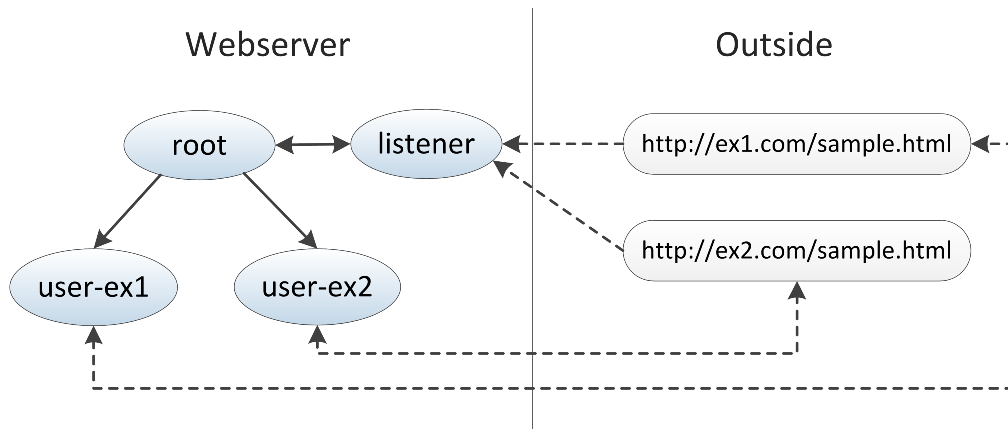


Figure 3.4.: Request processing with Privilege Separation: an unprivileged process (listener) is listening for incoming connections and forwards valid requests to a superuser process (root). This then forks request handling processes, that run under different users for each website.

Figure 3.4 shows, that also the delegation part differs from Apache's architecture: in order to meet the Privilege Separation principles and the principle of the Least Privilege,

request handling processes run with the privileges of the file being requested. In that way, in case of the execution of a CGI script or even in case of a failure, they can't access parts of the system they are not allowed to. For example, if a process responsibility is just to serve website A, it has no access to website B assuming that both websites are owned by different users, but on the same server.

We already mentioned earlier, that Apache performs pre-forking of request handling processes, for performance reasons [15]. However, in our architecture this methodology is not trivial, since the user, under which the request handling process is going to run, is not known in advance. Therefore, one would need to create several pre-forked processes for every system user that owns a website. Apache has in its standard configuration about 10 pre-forked processes, that stand ready for serving requests, and can serve any website. Let's assume a server hosts 50 small websites. Because of their smallness one can for example assume, that at most 10 percent of them are visited concurrently. In this case Apache could provide 2 processes per website. However, in order that our web-server can provide 2 processes per website, we would need 100 pre-forked processes for the 50 websites. This is because the websites being visited are not known in advance, and according to the Privilege Separation principles, one pre-forked process can serve just the website it is assigned to. In terms of system resource usage, it is a big difference, whether there are 10 or 100 server processes running concurrently. Therefore, we decided not to perform pre-forking, and accepted the delay at request time, i.e., the increase of the response time, by the time the operating system needs to execute the *fork* call [24].

In a nutshell, our architecture contains three types of processes:

- **Listener:** An **unprivileged** listener process, which is responsible for accepting new connections on the server-port (usually 80).
- **Root:** A **privileged** superuser process, which stays in background and does not communicate directly with the Internet. It performs administrative tasks such as process forking.
- **Request handler:** An **unprivileged** request handling process, which is responsible for processing an incoming request, i.e., generating a HTTP response message out of a request.

Figure 3.5.: UML component diagram, which illustrates how the three processes work together. The listener component is able to accept HTTP requests, via the HTTP interface. It forwards valid requests to the root component, which instantiates for every request a separate request handler.

3.2.2. Interprocess Communication

We described the general picture of the architecture so far, and the roles of the different types of processes. However, an important aspect is how the processes communicate with each other. We already described in section 2.2, that simple method calls between objects, maintained by different processes, are not possible, because one process can't access the memory space of another. Therefore Interprocess Communication is needed.

Let's initially step back and define the set up process of the server, i.e., how it will be instantiated. The server is just able to serve requests, when the listener and the root process are both running. Without a listener process the server is not reachable, and without the root process it cannot handle requests, since there is no privileged part, that can create request handling processes. On start-up, the server, therefore has to create both processes, i.e., one process can be *forked* out of the other. The listener process first starts as a superuser process, and performs some initialisations, like parsing the configuration files, creating listening sockets, etc. After this, it creates the root process, using the *fork* system call. This process automatically inherits a copy of all objects from the listener (side effect of *fork*, see section 2.2), and so, for example, both have access to the server configuration. After the fork, both processes go their own way (see Figure 3.6). The listener process binds itself to a network port, and gives up its privileges, since it is supposed to be unprivileged. When the listener accepts a new client connection, a TCP communication socket will be created, and the listener receives the request and validates its syntax.

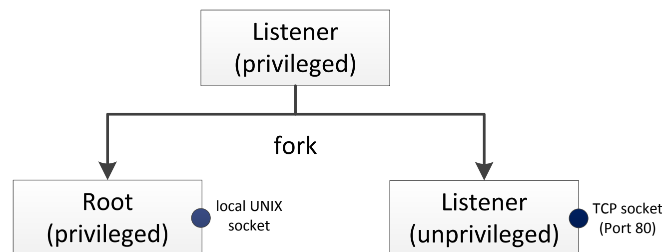


Figure 3.6.: Creation of the root process.

This leads us to the point, where Interprocess Communication becomes necessary, because the listener process needs a way to communicate valid requests to the root process. We already described earlier several approaches for IPC, such as Pipes, Message Queues, Shared Memory, Sockets, etc. Which approach was considered to be the best for our situation, will be clear soon. But consider first, what should happen, after the root process received a request from the listener. The fork methodology will be used again, and the root process creates a child process, that is going to handle the request. The child automatically inherits a copy of the request it has to process, so there is no extra IPC needed, between root process and its children. However, after the request was processed, the response must be streamed back to the client. Basically, the request

handling process would need to have access to the client communication socket from the listener. In this way, it could send everything directly back to the client. However, this socket was created by the listener process, after the root process was forked. Therefore, the root process did not inherit it, and so, neither the root process nor the request handling process have access to it. Furthermore, it is not possible to communicate sockets, i.e., file descriptors, between different processes. So, the only solution, to get the HTTP response back to the client, is to communicate it to the listener process via IPC. The listener has access to the clients communication socket, and can therefore forward the response to the client.

Summarizing, we are left up with two problems:

1. How to communicate the request data from the listener to the root process.

2. How to communicate the response data from the request handler to the listener.

We found a solution, that solves both problems using the same IPC method, namely sockets. On start-up, the root process creates a local Unix socket, and listens for incoming connections. This socket is not reachable from the Internet, but the listener process is able to connect to it.

After an incoming request on the server port and a successful validation by the listener process, this establishes a connection to the root process via the local Unix socket. The root process then creates a separate communication socket for the connection to the listener.

Now, there are two possibilities how to send the request data to the handling process: first, the idea was, that the root process receives the request data from the listener and then forks a new process, which automatically inherits the data, and handles the request. However, it would also be possible to let the request handler directly receive the request from the listener, to reduce the work load of the root process. This is possible, due to the fact, that the request handling process also inherits communication sockets from its parent. We considered this second solution as more efficient, and also more secure, since the root process should just perform tasks that need privileges [27].

So, in a nutshell, there is a separate connection for each client, between request handling process and listener, and between listener and client (see Appendix, Figures A.3, A.4, A.5). This solved our first problem, and one can reason, that it also solves the second problem, i.e., how to communicate the response data from the request handler to the listener, and then back to the client. Socket connections are bidirectional, therefore, the response can be sent using the same socket, where the request was received.

Concluding, the only responsibility of the root process is to accept connections from the listener, and create child processes, which inherit the connection. The root process immediately closes the connection to the listener afterwards, so it is not part of any further communication. It does also not have any communication channel to its children, so there is no way, that they could send messages to it and take it over. Data is transferred directly between listener and request handling process, i.e., two unprivileged processes.

Alternatives

The local Unix socket approach for Interprocess Communication was considered as most appropriate, because it provides separate communication channels for each client. Additionally it supports the Privilege Separation principles and the concept of the Least Privilege, because the root process' task is just to wait for incoming connections from the listener, and to fork request handling processes afterwards.

A shared memory or communication via files, were not considered as appropriate, because they would increase the complexity of the software. These approaches advantage communication between several processes, but in our case, we needed just to exchange data between two. Additionally, they are not ideal for data streams, since it is not easily detectable when new data is available. Furthermore, problems with concurrent accesses could occur, which would need synchronization and locking.

Also signals were not an option, since they can only be used as notifier for events, and not to exchange much data.

We did not want to use any middleware, because in that case, we would not have had fine grain control about the privileges of the different processes. Therefore RPC, Corba or similar approaches were not suitable for Interprocess Communication.

At the beginning of the project we made some experiments with pipes, since they are very similar to sockets, and also support the streaming of data. However, we were not able to use them, because a pipe allows just communication between a parent and its child process, and has to be created before the forking of the child process. We needed for every client a connection between the listener and the request handler, but this was not possible to achieve, since the request handler is a child of the root process, and not of the listener.

3.2.3. Class diagram

In this subsection we want to describe the architecture of our system, using UML (see Figure 3.7). However, consider, that this class diagram describes the logical structure of the system. Actually, there are three types of processes involved, and class instances are distributed among these processes. Interprocess Communication is used to exchange objects between processes.

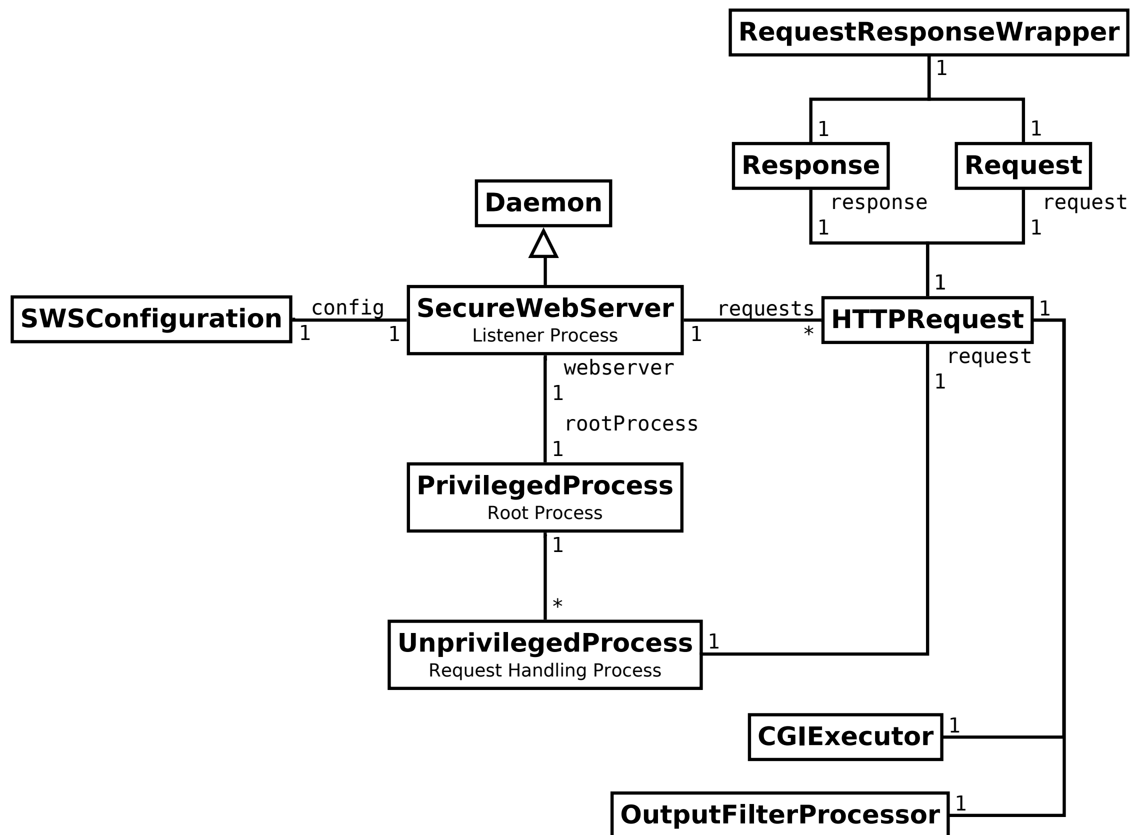


Figure 3.7.: UML Class Diagram. A more detailed version, containing the most important attributes and operations, can be found in the Appendix, Figures A.1 and A.2

Daemon

This class provides the base functionality of a daemon, so that the software can run as a system service, i.e., as a background process, detached from any terminal. The daemon can be controlled using the commands *start*, *stop* and *restart*.

SWSConfiguration

This class implements a configuration file parser. It is able to parse the main and all Virtualhost configuration files. This makes all the settings easily accessible to other classes. The parser is also responsible for validating the configuration files, and alerting the user if a directive contains syntax or semantic errors.

SecureWebServer

The SecureWebServer class is a daemon, and the main class for creating an instance of the web-server. On start-up it initiates the configuration file parsing process, and then takes over the role of the listener process. It maintains all the connections to the clients, and has for each client also a connection to the corresponding request handling process.

The class is responsible for the receiving, the parsing and the validation of requests. It communicates valid requests to request handling processes and responds to invalid ones with a Bad Request (400) error. It forwards response data, that it gets from a request handler, to the corresponding client. Note, that in some cases, the response is a local redirect response [18]. In this case it initiates the reprocessing of the request, using a different URI.

PrivilegedProcess

This class represents the root process of the architecture. It is responsible for accepting requests from the listener process, and creating unprivileged child processes, to which it delegates the handling of request.

UnprivilegedProcess

An UnprivilegedProcess instance is responsible for the request handling. It is created by the root process and is therefore initially privileged. It instantiates a HTTPRequest object, which receives the request from the listener. After that, it is able to determine the privileges of the requested resource, and gives up its privileges. Then it processes the request, sends back the response, and terminates.

HTTPRequest

This class represents a HTTP request. It provides operations for receiving, parsing and handling requests, and also for sending back responses. It can therefore be used by the listener and the request handling process. However, an instance of this class can not be communicated between processes, because it contains file descriptor attributes. We explained in earlier sections, that it is not possible to communicate file descriptors from one process to another. This problem is solved via the RequestResponseWrapper class: a HTTPRequest has two important member variables, which are Request and Response objects. They contain all the data of a request or a response, such as request method, URI, headers, body, etc. Both can be wrapped together (RequestResponseWrapper) and communicated between processes.

CGIExecutor

The CGIExecutor is responsible for executing CGI scripts. It aborts running CGI scripts that take too long, i.e., exceed the CGI time-out limit.

OutputFilterProcessor

This class applies Output Filters to the response's body data by pipe-lining. Similar to the CGIExecutor, it aborts running filter scripts, that exceed a time-out limit.

Request

This class contains all the attributes of a request, such as request method, URI, protocol, headers, body, file path of the requested resource, etc. For more details, see Appendix, Figure A.2.

Response

This class contains all the attributes of a response, such as status code, status message, protocol, headers, body, etc. For more details, see Appendix, Figure A.2.

RequestResponseWrapper

This class wraps together Request and Response objects, so that they can be communicated in one step between processes, and don't have to be sent separately.

4. Implementation

4.1. Software and libraries used

The Python programming language was selected to implement this project. The main reason for choosing Python (version 2.7.1+) was, that it is an object oriented language, and provides a large set of libraries and system interfaces [19], which are needed for developing server software. Python is portable and runs on many operating systems. This project was build and tested on Ubuntu, a Linux derivation based on Debian.

A GitHub repository was set up at *github.com/speer/sws*, so that the code is under version control and stored, i.e., backed, at a central place. Development and unit testing were performed in the Unix bash, using the VIM editor.

For implementing the web-server software, the following Python libraries have been used. They are all included in the standard distribution: *subprocess, multiprocessing, threading, socket, select, cPickle, urllib, logging, signal, pwd, grp, sys, atexit, stat, os, time, re* [19]. For performing unit testing with *PyUnit* [21], the *unittest* and *httplib* libraries were used.

In addition to all these standard libraries, two third party Python scripts have been used:

- **magic.py[1]:** Provides mime type detection of a file by analysing its content. This open source script is a wrapper around the *libmagic* file identification library, and is distributed under the PSF license [20].
- **daemon.py[2]:** Provides daemon functionality, so that the web-server can run as a background process. It is also open source and for public domain.

4.2. Asynchronous requests

It is very easy to create a synchronous web-server in Python. This handles requests sequentially, which means, that only one request can be processed at a time, and only one privileged process is involved. Python provides predefined libraries and classes, such as the *BaseHTTPRequestHandler*, which allow to implement such a basic web-server with just a few lines of code. Initially we used these libraries for our project, but soon we realised, that it is not the proper way to proceed. Consider for example a situation, where a user, with a slow internet connection, downloads a large file from a synchronous web-server. Other clients could not connect to the server for a long time,

[1] https://github.com/ahupp/python-magic
[2] http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python

because this is still busy sending back the large file. Such approach would result in a very bad performance, but beside that, also the Privilege Separation principles could not be applied, since there is just one web-server process.

We considered it therefore very important, that our web-server is able to process requests asynchronously. This means, that multiple requests can be handled at the same time. We described already in section 2.3, that in Apache, asynchronous requests are supported by several different multiprocessing modules, such as the *Prefork* module. By assigning each request immediately to a different process, Apache's listener process stays idle for accepting new connections.

Unfortunately, our web-server design makes asynchronous request handling more complex. Like in Apache's *Prefork* architecture, there is a separate request handling process for every request, so requests can be processed in parallel. However, the problem of our architecture is, that clients do not communicate directly with the request handling process, but with the listener, which mediates between request handler and client. The reason for this has already been explained in section 3.2.2, and is mainly because file descriptors can't be sent from one process to another. This means, that the listener must be able to handle all the connections to the clients, and for each client a connection to the corresponding request handling process. Additionally, it must be able to allow asynchronous communication on all connections.

If there would be just one client at a time, i.e., if the communication would be synchronous, this situation would not be a problem. The listener, basically running in an endless loop, could process requests in the following manner:

1. Accept connection from Browser 1
2. Receive request from Browser 1
3. Establish connection to root process
4. Send Browser 1's request to request handling process
5. Receive response from request handling process
6. Send response back to Browser 1
7. Close connections to Browser 1 and request handling process
8. Wait for new connection
9. Accept connection from Browser 2
10. etc.

A very important aspect in this case is, that the listener process blocks for some time at step 2 and at step 5. The *receive* call on a socket is usually blocking, because the process waits until some data arrives, and continues just after it received all data. Similar is it also with the *send* call in steps 4 and 6. Also sending data is blocking, until all data could be sent. This behaviour is beneficial for synchronous communication, however does not support asynchronism, since the listener process, while processing one request, can't accept new connections. We came up with three different solutions to this problem. All have been implemented, but finally we continued working with the solution we considered as most efficient.

Listener: fork

This approach introduces asynchronous communication in the listener process by the use of the *fork* system call. For every client connection, the listener process spawns an unprivileged child process, which inherits automatically the communication socket to the client, and is therefore able to receive the request data. Then this child establishes a connection to the root process and sends the request to the handling process. Afterwards in can wait for the response, and then send it back to the client. At the end the child process terminates.

Note, that here the blocking *receive* or *send* calls do not cause any problems, since they affect just the child process, and not the listener, which is at any point able to accept new connections. The listener's work load was so drastically reduced, i.e., split up among it's child processes. Although each child process communicates synchronously, the architecture as a whole achieves asynchronism, and therefore, the web-server is able to handle multiple requests concurrently.

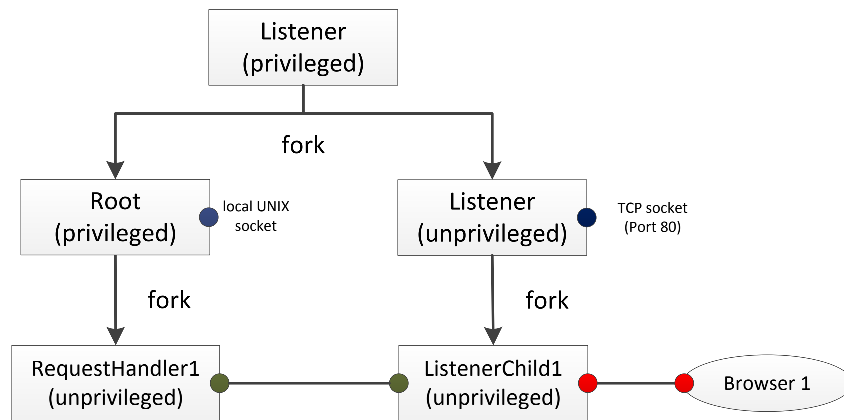


Figure 4.1.: Achieving asynchronous communication in the listener process using *fork*.

However, in the long run, we were not satisfied with this approach, because the web-server needed to create two new processes for every request: the child of the listener and the child of the root process, i.e., the request handler. We already mentioned earlier that a *fork* system call increases the response time, by the time the operating system needs for its execution. Therefore we implemented another solution, which just needs to *fork* once - basically the request handling process - and reduces so the delay time of the response.

Listener: select

In order achieve asynchronism, without forking the listener process, we had to solve the problem of the blocking *receive* and *send* calls. Sockets can operate in non-blocking mode. In this case, if the *receive* call does not immediately get any data, or the *send* call does not immediately send data, the program continues executing and does not wait [23]. However, setting the sockets to non-blocking means, that after a *receive* call, one cannot be sure any more that there is any data available. This further implies, that one cannot continue execution like with a blocking socket. This changes the structure of the software completely, since one needs to detect, when a socket is ready to receive or send data.

The *select* system call provides a solution to this problem: it monitors multiple file descriptors, i.e., sockets, and waits until at least one becomes ready for an I/O operation [22]. The call returns three lists of sockets, which are ready to send, receive or produced an error. This provides a mechanism to detect, when to can call *send* or *receive* and for which socket.

We changed the implementation of the listener, and monitored every communication socket using *select*. There are three types of sockets to distinguish: connections to the client, connections to the root process, and the server socket (ex. port 80), which is the entry point for new connections.

So, the endless loop, in the implementation of the listener, is left with just one blocking call, namely *select*. This call returns as soon as one or more sockets are ready, and one can iterate over them and call *send* or *receive* respectively. These calls won't block, so they return immediately after having completed their job. After having iterated over all ready sockets, the listener calls *select* again and waits for the next sockets being ready, etc.

For every client, the listener mainly executes the following non-blocking steps, as soon as the corresponding socket is ready for the I/O operation:

1. Server socket: accept a new connections from a client.
2. Client communication socket: receive the request from the client.
3. Request handler communication socket: send the client's request to the handling process.
4. Request handler communication socket: receive the handling process' response.
5. Client communication socket: send the response to the client.

Basically, the listener is now able to serve multiple requests concurrently, without having to make use of the *fork* system call twice. Performance tests made using ApacheBenchmark [12] showed, that for small files, the select approach reduced the response time by up to 50 percent. For big files the delay time of the *fork* call was not a decisive factor, since they generally need longer for being transported, especially over a network.

Listener: epoll

This third approach is based on the same principles as the *select* approach. However, instead of the *select* call, the *epoll* system call is used for event notification. Note, that *epoll* is just available on Linux systems with a kernel version larger than 2.5.44 [22].

In order to explain the difference between the two system calls, we first want to explain the concept of a file descriptor. File descriptors are needed for accessing files, i.e., for performing read and write operations. They are valid in the scope of a process, and on Unix systems represented by integer numbers. So is the file descriptor for standard input usually represented by the number 0, for standard output by 1 and for standard error by 2. Sockets are types of file descriptors, and get therefore also unique numbers. Note, that I/O operations on sockets are called receive and send, instead of read and write, but basically have the same meaning.

Both, the *select* and the *epoll* system calls, monitor multiple file descriptors, until they are ready for an I/O operation. The difference lies in their implementation and their time complexity [22]. The complexity is linear in both cases, but *epoll*'s is slightly better:

1. Epoll: $O(number\ of\ file\ descriptors)$
2. Select: $O(highest\ file\ descriptor)$

Epoll just considers the file descriptors of interest, i.e., those that are monitored [22]. So, if there are 10 concurrent requests, epoll's time complexity will be $O(21)$. This is because there are 10 connections between clients and listener, 10 connections between listener and request handlers, and the server socket that is always being monitored, in order to accept new connections.

Select's complexity depends on all file descriptors, the process obtains, independently whether they are monitored or not. *Select* initially considers all file descriptors that are available, then it flags those that are monitored, and lastly it scans all of them linearly, in order to determine which flagged ones are ready [22]. Therefore, if a process obtains many file descriptors that are not monitored, *select*'s performance will be worse than *epoll*'s. Our previous example, with 10 concurrent requests, therefore has at least a time complexity of $O(24)$, because one has to add the descriptors for standard input, output and error, which are available in any process. Consider, that this number could be even higher, since there might also be other file descriptors for log files, configuration files, etc.

However, in a nutshell, it does not make much difference in our case, whether to use *select* or *epoll*, since the listener process does not obtain many additional file descriptors, what would make *select* much slower. Also, we could not measure any difference in the web-server's response time, between the two approaches. We considered *epoll* to be the solution to go on with, since in theory it should be slightly faster.

4.3. Request processing

In this section we want to explain step by step the tasks that are executed by our server, in order to generate a HTTP response message, out of a HTTP request.

4.3.1. Receiving the request

We already stated earlier, that the listener process is responsible for receiving a HTTP request from the client. This is done by instantiating a *HTTPRequest* object, and by calling the *receiveRequestFromClient* operation. Consider, that the data communicated between processes is streamed. Therefore, one cannot be sure, that all the data arrives at the same time, but one might need to call *receive* several times. This is handled by the *HTTPRequest* object, which has mechanisms to detect when it got all the data, and should start parsing the request.

While parsing the request, *Request* and *Response* objects get initialised. These are part of the *HTTPRequest* object, and contain the information of the HTTP request message. The *Response* object is initialised with status code 200, which means success. If there is any error in parsing the request, its status code will change, and so, the listener process knows whether it should forward the request to the handling process, or respond directly with an error to the client.

If the request is valid, the listener establishes a connection to the root process, which spawns a request handling process (see section 3.2.2). After the connection to the handling process has been established, the listener packs the *Request* and *Response* objects into a *RequestResponseWrapper* object, which gets encoded using the *cPickle* library. In this way it can be sent to the request handling process.

The request handling process instantiates also a *HTTPRequest* object, and uses the *receiveRequestFromListener* operation, to receive the encoded *RequestResponseWrapper* object from the listener. After it was fully received, it gets decoded, and the *HTTPRequest*'s member variables get initialised with the content of the wrapper object.

4.3.2. Processing the request

The *HTTPRequest* object has an operation called *process*, which is responsible for performing all the request processing, and also sends the response back to the listener. All major tasks that are executed by this method will be described here:

Initially it checks whether the requested resource is located inside the document root directory. This directory defines the environment to be published by the server. Consider, that there are several Virtualhosts, and each has its own document root directory. The determination of the matching Virtualhost did already take place during the parsing process. It is a very important security aspect, to guarantee that users can't access files, which are stored outside the document root directory. In such a situation, the request handler stops further processing, and sends back a Forbidden error (403).

Next, the request handling process determines some properties of the *HTTPRequest* object:

1. **CGI script:** Does the configuration specify a CGI handler for this request URI?
2. **Output filter:** Does the configuration specify Output Filters for this request?
3. **DirectoryIndex:** Is the request URI a directory, and are there eventually some *DirectoryIndex* files that match?
4. **404 - Not Found:** Does the requested resource exist?
5. **403 - Forbidden:** Is the user allowed to access the resource?

Depending on these properties, the request handler either delivers a static document, executes a CGI script, or sends back an error document.

However, before proceeding with this, the request handling process needs to give up its privileges, in order to comply with the principle of the Least Privilege [27]. Up until this point, the request handling process was still privileged, because earlier it was not possible to identify the requested resource on the file system. Now, knowing all the request's properties, the request handling process can detect the owner and the group of the requested file, and change its own user and group id respectively. This is done by the following Python code:

```python
# detection of the file's properties
st = os.stat(self.request.filepath)

# if file is owned by root,
# try to access is as default user for security reasons
if st.st_uid == 0:
        # change process' privileges to: default user
        os.setgid(self.config.configurations['group'])
        os.setuid(self.config.configurations['user'])
else:
        # change process' privileges to: file owner user
        os.setgid(st.st_gid)
        os.setuid(st.st_uid)
```

Note, that this removal of privileges can only be done once and does only work, because the request handler initially is a superuser process. After being unprivileged, the process can't gain privileges any more. This is the main reason, why the web-server is considered to be secure.

Static document delivery

First, the content type and the size of the requested file will be determined. These values are represented by two header fields, that are supposed to be set in the HTTP response, so that the browser is able to display the document correctly. Then the rest of the header part of the HTTP response message can be assembled, except the body, which will be the content of the requested document. Now the streaming functionality of the server comes into play: data is read from the requested file block wise, and each block of data is immediately sent to the listener (*flushResponseToListener*), which immediately forwards

it to the client (*flushResponseToClient*). Note, that the first time this operation is called, also the previously generated header part will be prepended to the HTTP response. This streaming approach was more difficult to implement, than delivering the whole HTTP response in one go, but it increased the efficiency of the web-server. Initially, requested documents were gathered in memory before being delivered to the client. This approach led to high memory usage, especially when big files were requested. With streaming, data will not be accumulated in memory, and out of memory errors can be avoided.

If there would occur any error, while accessing the document, the server will respond with a Internal Server Error (500). However, this is just possible, if no data has already been sent to the client. If the HTTP response's status line and the header fields have already been streamed out, there is no way for the web-server to notify the client that an error occurred. In this case, the connection will simply be closed.

CGI script execution

Initially, the script file is verified for being executable, and the CGI variables are prepared. Then an instance of the *CGIExecutor* class executes the script using the *subprocess* library, actually the *Popen* function. It creates a separate process for the script, which has the same privileges as the request handling process. Additionally, it allows to pass the CGI variables, as environment variables to the script, and the request's body data to the script's standard input file descriptor. During the CGI execution, any data, that will be produced on the script's standard output file descriptor, will be streamed to the listener, using the *flushResponseToListener* operation. Anything, produced on the standard error file descriptor, will be stored in the error log file.

We defined in the requirements, that running CGI scripts should be aborted after a given time-out. In order to realise this, the request handling process initially creates a separate thread, which invokes the CGI script using the *Popen* function (see Appendix, Figure A.6). This thread also streams the script's output to the listener. The main thread is responsible to monitor the execution: in case the time-out is reached, it aborts the CGI script and also the other thread.

In case of any error during the execution of the CGI script, forced or unforced, the server will return an Internal Server Error (500).

Error documents

Error documents can be defined in configuration files for different status codes. These documents can either be static or dynamic, i.e., CGI scripts. If an error needs to be returned in the HTTP response, the request handling process looks up which error document to use, and then makes use of the two functionalities, that have been described above: either delivers a static document or executes a CGI script. If there occurs an error during this process, it simply returns a predefined error message, instead of the error document.

Note, that error documents are only provided, if no data has already been streamed to the client. Otherwise the connections will be just closed.

4.3.3. Sending the response

We already explained earlier how the response will be streamed to the listener, which then forwards it to the client. Data between listener and client will be sent in the format of a HTTP 1.1 response message. However, between request handling process and listener, the *RequestResponseWrapper* class is used.

Now, one might think, why this *RequestResponseWrapper* is needed. Why don't simply send the *Request* object from listener to request handler at request time, and the *Response* object from handler to listener at response time? The reason for this is, that in some cases, a response can cause another request: this is called a CGI local redirect [18]. In this case, the listener, instead of forwarding the response to the client, establishes again a connection to the root process, and causes the reprocessing of the request, but with a different URI. All the data of the old request, has to be available also in the new request, therefore this wrapper class is needed for communicating *Request* and *Response* objects in both directions.

However, this reprocessing could lead to endless redirections, i.e., if a script redirects to itself. We created a configuration directive, which specifies the maximum number of allowed redirections, after the server responds with an Internal Server Error (500).

Output Filters

The application of Output Filters, i.e., scripts that modify the response body, is realised inside the *flushResponseToListener* operation. This is used generally for streaming data to the listener. However, as soon as a request matches an Output Filter, the streaming functionality is disabled. The reason for this is to guarantee, that any Output Filter does it's processing properly. Consider for example a filter, that does text replacements and should replace the word "hello", with "good morning". If the server would stream the data, it could happen that the filter gets the string "hel" in the first stream, and "lo" in the second. It can not find a match for "hello", and therefore does not behave in the desired way. So, all filter scripts must get the response body data in one go. Of course, the disadvantage of this approach is, that if the document is big, it leads to higher memory usage.

Output Filters are executed one after the other, by the request handling process, using the *OutputFilterProcessor* class. Initially the response body data will be provided on the standard input file descriptor of the first script. Using pipe-lining, the scripts output will be provided as input to the script next in the filter chain, etc. The output of the last script will form the final response body of the HTTP message.

Python's *Popen* function is used for executing filter scripts. Therefore, they run with the privileges of the request handling process, which means that they are unprivileged. Like CGI scripts, they will be aborted by the handling process, if their execution time exceeds the predefined time-out value.

5. Evaluation

5.1. Research hypothesis

The main research hypothesis that motivated our project is that the application of the Privilege Separation principles [25] and the concept of the Least Privilege [27] make a web-server more secure. In order to prove this, we compared two different web-servers: we installed the Apache server and the web-server that has been created in this project. On both servers we set up the same web-sites, and gave them just the privileges that they really need for being served by each server. We stated the following hypothesis:

H: Privilege Separation makes a web-server more secure.
We expected, that our web-server is more secure than a standard configuration of Apache, because it performs Privilege Separation. Therefore, a CGI script of one website should not be able to affect other websites, running on the same server. Adversaries should also not be able to take over the server system from the Internet, by exploiting a bug.

5.2. Evaluation strategy

In order to test the above mentioned hypothesis, we first set up an evaluation environment: we installed the Apache server and our web-server. On both servers we created two Virtualhosts, named site1 and site2. The corresponding sites contained exactly the same files on each server. Apache was set up the following way:

site1: Document root directory: apache/site1 (Owner: www-data, Privileges: 500)
CGI script: site1script.php (Owner: www-data, Privileges: 500)

site2: Document root directory: apache/site2 (Owner: www-data, Privileges: 700)
CGI script: site2script.php (Owner: www-data, Privileges: 500)

Consider, that the Apache user (*www-data*) needs at least read privileges for the files it has to serve. This requirement is satisfied, so Apache is able to serve both Virtualhosts. Additionally all the privileges are kept minimal, except the document root directory of site2 grants also write privileges to Apache. This is not uncommon, since site2 could possibly need to create some local files (for example like a picture upload application). The configuration of our web-server was like this:

site1: Document root directory: sws/site1 (Owner: site1, Privileges: 500)
CGI script: site1script.php (Owner: site1, Privileges: 500)

site2: Document root directory: sws/site2 (Owner: site2, Privileges: 700)
CGI script: site2script.php (Owner: site2, Privileges: 500)

Consider that there are individual users for every website, since our web-server does not need a common user for all Virtualhosts. Like in the Apache configuration, the privileges are kept minimal, except for site2, where the user has also write privileges.

We assumed that both websites are owned by malicious users, which try to get access to other websites of the server, using PHP scripts:

site1: Tries to delete all the files from site2. The code can be found in Appendix C.1.

site2: Tries to copy all files from site1 to its own document root directory. This is the reason, why site2's document root directory grants write privileges (700) to the web-server user. The code can be found in Appendix C.2.

The actions of both scripts are very dangerous, because they try to get access to protected data, or try to delete other website's content. We considered this as an appropriate example to test, which web-server is secure, i.e., whether our web-server performs Privilege Separation correctly.

We expected that Apache would execute both scripts with no warnings, and therefore support both adversaries. Our web-server in contrast, should deny the script's dangerous actions.

Consider, that this project was build also along a second security aspect: adversaries should not be able to take over the system from the Internet. The evaluation of this aspect is not trivial, since it is hardly possible to exploit a bug in a software, such as the Apache server, and analyse its behaviour. However, we know, that Apache's architecture has a privileged listener process, which means, that clients establish connections to a superuser process. In our architecture the listener process, which represents the interface to the World Wide Web, is unprivileged. Therefore, in case this process is taken over by an attacker, for example using techniques such as code injection, it cannot harm the system with superuser privileges [26]. Under Unix there exist several tools to test the privileges of a process, such as *top*, *ps* and *netstat*.

Beyond testing the listener's privileges, we also wanted to analyse how it behaves in case of a takeover. Since our system does not have known bugs, we temporarily introduced two different errors, and exploited them separately for evaluation purposes:

bug1: Using this bug we wanted to break and interrupt the request parsing. This is generally a critical stage, since one can't predict any type of input that will be provided. Badly formatted messages could possibly cause unexpected behaviour, although the HTTP parser is designed to be very tolerant, and also performs syntax checking. In case a message does not meet the HTTP specification, it responds generally with a Bad Request (400) error to the client.
However, this bug forces the parser, which is executed by the listener process, to raise an exception if it gets a HTTP header field, named bug1. Theoretically, the exception should abort the parsing process, and the listener should not forward

the invalid request to the root process, for security reasons. Even though an exception occurred, the server is supposed to stay stable.

bug2: This bug tries to verify, whether the listener process is able to harm the system. It tests the consequences of code, that will be injected into the listener process. For this we introduced a second HTTP request header field, named bug2, which forces the listener to execute the command, specified in the header's value. So, if one would set a header such as *bug2: rm -rf /*, the server will try to erase the whole file system. Since the listener is assumed to run as an unprivileged process, a command, specified in this header, should not harm the system.

We also performed unit testing using *PyUnit*, in order to reduce the number of bugs in our web-server. The less known bugs there are in software, the more secure it can be considered.

5.2.1. Additional aspects

Performance

We also wanted to compare the performance of Apache and our web-server. Therefore we performed speed tests using the ApacheBenchmark tool [12]. It allows to fire multiple concurrent or sequential requests to a given URL, and measures the time the server needs for responding. In order to avoid network delays, the tests have been conducted locally.

In this case Apache was our clear favourite, mainly because of its architecture. Apache's request handling processes communicate directly with the clients and no data has to be passed through the listener process. Additionally, Apache's *Prefork* module spawns processes in advance, so there is no delay at request time because of process creation [15]. Another reason is, that Apache is programmed in C, and not in an interpreted language such as Python.

We set up three different types of resources, on both, our web-server and Apache: a small static HTML website, a medium size JPG picture and a PHP script. Using ApacheBenchmark we fired for every resource 100 requests to the servers, and measured some statistics. Out of those 100 requests, always 10 have been sent concurrently.

Operability

Lastly, we also wanted to evaluate the software according to its functionalities. It should be able to deliver any type of static and dynamic document, i.e., images, HTML files, binaries, PHP scripts, bash scripts, perl scripts, etc. Additionally it should work for small documents as well as for large ones. In order to perform these tests, several document types and file sizes have been tested on today's major browsers, and also on other HTTP clients.

In order to test the functioning of complex dynamic websites, we set up CMS systems such as Drupal and MediaWiki. For evaluating the Output Filter functionality we created several example scripts, like a text replacement filter, a gzip compression filter, and a filter that automatically includes a header and a footer part in any page of a website.

5.3. Evaluation results

We executed both scripts, i.e., site1 and site2, on both, Apache and our web-server, and the results were as we expected. Our web-server denied the access to web-sites owned by other users, and therefore performed Privilege Separation correctly:

```
PHP Warning:  opendir(/home/stefan/sws/docs/apache/site1/):
failed to open dir: Permission denied in /home/stefan/sws/docs/
sws/site2/site2script.php on line 3

PHP Warning:  opendir(/home/stefan/sws/docs/apache/site2/):
failed to open dir: Permission denied in /home/stefan/sws/docs/
sws/site1/site1script.php on line 5
```

Apache executed both scripts successfully: script2 was able to copy all files from site1 to its local directory, and script1 deleted all files from site2. This proves, that Apache does not separate the privileges of different websites.

In order to test the privileges of the listener processes of both servers, we used first the *netstat* tool to detect their process ids:

```
netstat -antp
tcp  0  0 0.0.0.0:80     0.0.0.0:*   LISTEN  28739/python
tcp  0  0 0.0.0.0:8080   0.0.0.0:*   LISTEN  27797/apache2
```

In our environment, Apache's listener process has therefore process id 27797 and our server 28739. Then we used the *ps* Unix tool, to detect the users, under which these processes are running.

```
ps aux | grep sws
stefan  28739  /usr/bin/python -B ./sws restart
root    28740  /usr/bin/python -B ./sws restart
site3   28742  /usr/bin/python -B ./sws restart

ps aux | grep apache
root      27797  /usr/sbin/apache2 -k start
www-data  28195  /usr/sbin/apache2 -k start
www-data  28196  /usr/sbin/apache2 -k start
```

This shows, that Apache's listener process, with id 27797 is running as *root* user, and is therefore privileged. Our web-server's listener with process id 28739 runs as an unprivileged user, called *stefan*. This proves, that our web-server can be considered as secure, since no privileged listener process is exposed directly to the Internet.

The root process with id 28740 is the privileged process, which stays in the background of our web-server. The unprivileged *site3* process, is a request handling process of our web-server, which is currently processing a request. Several *www-data* processes, that are shown by *ps*, are pre-forked request handling processes of Apache.

As already mentioned, we introduced temporarily two different bugs in the listener process of our server, in order test its behaviour in case it would be taken over by an attacker.

```
if key == 'bug1':
        raise Exception
if key == 'bug2':
        subprocess.Popen(value.split())
```

Initially we tested bug1 by establishing a TCP connection to our server, via telnet, and by issuing the following request:

```
GET / HTTP/1.1
host: localhost
bug1: 1
```

The request has a valid HTTP syntax, however, because of the bug in the request parser, the listener process raised an exception. Note, that the exception was not caught explicitly by the server, in order to obtain a realistic behaviour:

- The server closed the connection to the client immediately, without giving any response message, even not a Bad Request (400) error.
- The exception did not have any impact on the server's stability, i.e., it was still responding afterwards, and could handle other requests.
- In the error log file, the following message appeared: *Communication error at file descriptor 22*.
- We monitored the system's processes via *top* and *ps* during this experiment, and detected, that no request handling process was created, i.e., the listener process did not forward the request to the root process.

Therefore we can state, that even though an exception occurred while parsing a request, the server remained stable, and did not threat the system's security.

In order to exploit bug2 we first created a file, named *testfile*, in the root user's home directory. Access was just granted to a privileged process: Owner: root, Privileges: 700. Then we issued the following request via telnet:

```
GET / HTTP/1.1
host: localhost
bug2: rm -rf /root/testfile
```

Because of the listener process' vulnerability to code injection, this request caused the execution of the *rm* command, specified in the bug2 header. However, we discovered, that the stated file had not been deleted. This means, that the listener process was not able to access it, i.e., has no privileged access to the system. Note, that we also tried a counterexample, to prove the correct functionality of the code injection bug: changing the privileges to 777, caused the removal of the file.

Because of the web-server's behaviour in these two situations, we can state, that a bug in the listener process, which might lead to a takeover, does not give an adversary the possibility to gain superuser privileges on the system. This is mainly because, invalid requests do not get forwarded to the root process, and the listener process is unprivileged. Also the HTTP response does not provide a way for an attacker to obtain superuser privileges, since it is sent directly from request handling process to the listener, without involving the root process. This proves, that the root process is completely isolated, and cannot be overtaken because of badly formatted messages.

We mentioned earlier, that we also performed unit testing, for avoiding bugs in our software. The final run of the test suite showed, that no errors have been detected (see Appendix, Figure A.7).

Conclusions

Because our web-server has no known bugs, performs Privilege Separation correctly, has an unprivileged listener process and an isolated root process, it can be considered as secure. We can therefore accept our main hypothesis, stated in section 5.1: Privilege Separation makes a web-server more secure, i.e., our web-server is more secure than a standard configuration of the Apache server.

However we want to mention a very important aspect: one should not reason based on these results, that Apache is a insecure web-server. We considered security just with respect to Privilege Separation, since this is the focus point of our research. There are several modules for Apache, which make it more secure, such as *suExec* for example [10]. Using this module, also Apache would deny the execution of the malicious scripts of site1 and site2. Additionally, a privileged listener process is not a severe problem for Apache, since it is a very stable web-server, tested by a community of many people. However, if one would find a bug in Apache's listener and could exploit it, he could theoretically gain superuser privileges on the system. Regarding this point, our web-server is definitively more secure.

5.3.1. Additional aspects

Performance

The benchmark confirms our assumptions, that Apache's performance is much better. For every benchmark test we measured four different statistics. All of them represent the average values out of 100 requests:

- **Data transferred [byte]:** is the sum of the sizes of the HTTP request and the response.
- **Request per second [absolute value]:** represents the number of requests, the server can theoretically complete in a second. Note, that this value is calculated based on the time per request.
- **Time per request [millisecond]:** is the time, the server needs to complete a single request.

- **Transfer rate [kilobyte / second]:** specifies the average transfer rate of sending and receiving data. Note, that the tests were performed locally.

Server	Measure	HTML	JPG	CGI:php
SWS	Data transferred [byte]	363	59349	49588
	Requests per second	237	134	36
	Time per request [ms]	4.2	7.4	27.7
	Transfer rate [kb/s]	84	7791	1745
Apache	Data transferred [byte]	488	59496	51663
	Requests per second	3274	2752	1082
	Time per request [ms]	0.3	0.4	0.9
	Transfer rate [kb/s]	1560	159944	54836

Table 5.1.: Results of the benchmark test: average values of 10 times 10 concurrent requests.

The average time per request, is the most interesting value, since the others mostly depend on it. According to this results, Apache is generally speaking more than 10 times faster than our server. We came up with several different reasons for this huge performance difference:

- Apache is written in C, which is compiled to machine code and therefore executes faster than Python, which interprets byte code.
- Apache performs pre-forking, so it can serve requests without performing any *fork* operation. We already mentioned earlier, that forking processes causes delays, because it needs a system call. Our server needs to create at least one process for every single request.
- Our server has to communicate all the request and response data through the listener process, while Apache's request handling process can communicate directly with the client.
- Apache comes with native modules, that are responsible for executing PHP scripts. In our architecture a PHP script is treated like a CGI script, which needs a further process for being executed, i.e., a child of the request handling process. Therefore, two *fork* calls are needed for a CGI request, while Apache does not need to spawn any process at request time.
- During development we detected, that our approach for mime-type detection is quite slow: we use *libmagic*, which analyses the content of a file. Apache maps file-name extensions to mime-types, which just needs a list of mappings. Therefore, Apache can determine the mime-type of a file much faster.

Considering the benchmark results, we think that the performance of our web-server is not excellent but acceptable. Furthermore, the benchmarking tests have been performed locally, and networking delays have not been considered. Concurrent local requests stress our server much more, because of the *epoll* implementation: all the request and response

data is passed through a single listener process. Accessing the server just locally means, that sockets are sooner ready to send or receive data. This increases the current work load of the listener, because it decreases its idle time, and slows down so the web-server.

Note, that already while developing the system we tried out another Python interpreter, called PyPy, which basically is a Just-In-Time compiler [4]. This achieves a speed up, after a process runs for a while, because it creates executable machine code, instead of interpretable byte code. Initially we expected it can beat the standard Python, but it was double as slow. This was most likely due to the fact, that we often create new processes, and they have a very short life time. Therefore a process cannot gain the advantages of the speed up, that would be achieved after a while, because of the Just-In-Time compilation.

Operability

The following matrix shows, that all tested document types were handled correctly, by all tested HTTP clients. Also the Drupal and MediaWiki CMS systems, based on PHP, and all the three types of tested Output Filters, were working properly. Even functions such as PHP session management or Drupal's picture upload, did not point out any problems. Therefore we can say, that our web-server supports the most important parts of the current HTTP and CGI standards, and can therefore be used in practice.

	Firefox	Chrome	MS IE	Safari	Opera	Lynx	Wget	Telnet
HTML	ok	ok	ok	ok	ok	ok	ok	ok
JPG	ok	ok	ok	ok	ok	ok	ok	ok
Binary	ok	ok	ok	ok	ok	ok	ok	ok
CGI:php	ok	ok	ok	ok	ok	ok	ok	ok
CGI:bash	ok	ok	ok	ok	ok	ok	ok	ok
CGI:perl	ok	ok	ok	ok	ok	ok	ok	ok
Error:404	ok	ok	ok	ok	ok	ok	ok	ok
MediaWiki	ok	ok	ok	ok	ok	ok	ok	ok
Drupal	ok	ok	ok	ok	ok	ok	ok	ok
Replace Filter	ok	ok	ok	ok	ok	ok	ok	ok
Gzip Filter	ok	ok	ok	ok	ok	ok	ok	ok
Include Filter	ok	ok	ok	ok	ok	ok	ok	ok

Table 5.2.: Tested document types and HTTP clients, i.e., browsers.

6. Conclusions

6.1. Summary

In this project we wanted to test if Privilege Separation can make a web-server more secure. We created a fully functional web-server, based on the Privilege Separation principles [25] and the concept of the Least Privilege [27]. Therefore, a website should not have access to another website, that is hosted on the same web-server. Furthermore, a privileged process should not be directly exposed to the Internet.

Our web-server is able to deliver static and dynamic documents, basically CGI scripts. In addition it provides a set of other functionality, which is often needed by today's websites, such as Output Filters. Everything that was initially planned, was also realised.

In order to evaluate the software we created, we compared it with nowadays most used web-server: Apache. We mainly focused on the security aspects, i.e., whether the server performs Privilege Separation correctly. In addition we also tested its performance, in order to find out, whether it can be used in practice. We set up the same websites and scripts on both web-servers and discovered, that our server performs Privilege Separation correctly, and behaves in a more secure way than Apache. However, Apache's performance, i.e., response time, was much better.

The biggest challenge faced during this project was the realisation of the asynchronous Interprocess Communication. The support for several types of processes, privileged or unprivileged, was implemented soon, but they must also communicate among each other in an efficient way. We came up with several approaches and finally picked the one we considered as optimal, where all communication is done via sockets, and the listener process uses the *epoll* system call, for being able to handle multiple requests concurrently.

In conclusion, our web-server is lightweight and secure, with practical functionality and decent performance. It can be used in environments, where security is necessarily important. An area of application could be for example in data centres, where hosting providers want to provide cheap web-space to clients, and don't want the web-server to consume many system resources. Security is a very important aspect, since customers don't know each other, and therefore file accesses across websites should be strictly forbidden.

Currently, our system does work just on Unix systems, but since Python runs on many different operating systems, it can be ported with minor changes also to other platforms.

6.2. Future work

In the future, this project could be extended and optimised in various directions, such as performance, functionality or security.

We considered just the most important parts of the extensive HTTP 1.1 specification. Concepts such as persistent connections, transfer encodings, or user authentication have not been implemented at all. Persistent connections could possibly increase the performance of the web-server, because they allow a browser to send more than one request through a single connection. This could be an interesting field to focus on, since it needs some thinking, how the Privilege Separation principles can be applied in that case.

The weakest point of our implementation is the web-server's performance. We tried to make the request processing as fast as possible, however, because of the short project period, we mainly focussed on Privilege Separation. It would be very interesting to investigate, how to improve the performance. Beside keep-alive connections, also a pre-fork mechanism could bring some optimisation. However, it is not elementary to perform pre-forking, because the processes' privileges are not known in advance. These depend on the requested file, which is just known at request time. A possible solution to this could be, setting the privileges at request time, but forking privileged processes in advance. The counterpart of this approach is, that Interprocess Communication gets more complicated.

Performance optimisations could also be achieved by developing modules, which natively support dynamic scripting languages like PHP, or which support fastCGI. Also a more efficient mime-type detection algorithm could increase the server's performance.

Other future work could extend the web-server's functionality: we implemented just a limited subset of Apache's range of functions. Very interesting for nowadays websites could be for example a URL rewriter, or the support for *htaccess* files.

There is also an aspect regarding security, which could be optimised. CGI scripts of our server are executed by their owner-user. However, if this user has access to system folders, such as */etc* or */bin*, also the script has access to these folders. This could lead to security problems, since the script itself is not locked into the document root directory, and therefore has unprivileged access to the whole system. So, it is for example possible, that the script can read configuration files in the */etc* folder, which have set the privileges to 755, and possibly contain passwords or other confidential information. An approach to solve this problem could possibly be, restricting the script's execution environment to the document root directory.

References

[1] Bradford L. Barrett. Webalizer, 2012. URL http://http://www.webalizer.org/. (accessed August 15, 2012).

[2] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[3] CERN. How the web began, 2008. URL http://public.web.cern.ch/public/en/about/WebStory-en.html. (accessed August 2, 2012).

[4] PyPy Community. Pypy, 2012. URL http://pypy.org/. (accessed August 21, 2012).

[5] Internet Engineering Task Force. Requirements for internet hosts - communication layers, October 1989. URL http://tools.ietf.org/html/rfc1122. (accessed August 10, 2012).

[6] Apache Software Foundation. Request processing in apache, 2008. URL http://www.apachetutor.org/dev/request. (accessed August 11, 2012).

[7] Apache Software Foundation. Httpd privilege separation, September 2009. URL http://wiki.apache.org/httpd/PrivilegeSeparation. (accessed April 16, 2012).

[8] Apache Software Foundation. Apache 2.0 core features - directory directive, 2011. URL http://httpd.apache.org/docs/2.0/en/mod/core.html#directory. (accessed April 19, 2012).

[9] Apache Software Foundation. Apache 2.0 filter documentation, 2011. URL http://httpd.apache.org/docs/2.0/en/filter.html. (accessed April 19, 2012).

[10] Apache Software Foundation. Apache 2.0 suexec support, 2011. URL http://httpd.apache.org/docs/2.0/en/suexec.html. (accessed April 16, 2012).

[11] Apache Software Foundation. Apache 2.0 virtual host documentation, 2011. URL http://httpd.apache.org/docs/2.0/en/vhosts. (accessed April 17, 2012).

[12] Apache Software Foundation. ab - apache http server benchmarking tool, 2012. URL http://httpd.apache.org/docs/2.2/programs/ab.html. (accessed August 21, 2012).

[13] Apache Software Foundation. Apache log files, 2012. URL http://httpd.apache.org/docs/2.2/logs.html. (accessed August 15, 2012).

[14] Apache Software Foundation. Multi-processing modules (mpms), 2012. URL http://httpd.apache.org/docs/2.2/en/mpm.html. (accessed August 10, 2012).

[15] Apache Software Foundation. Apache mpm prefork, 2012. URL http://httpd.apache.org/docs/2.2/en/mod/prefork.html. (accessed August 10, 2012).

[16] Apache Software Foundation. Apache http server, April 2012. URL http://projects.apache.org/projects/http_server.html. (accessed August 2, 2012).

[17] Apache Software Foundation, D. Robinson, and K. Coar. The common gateway interface (cgi) version 1.1, October 2004. URL http://www.ietf.org/rfc/rfc3875. (accessed August 10, 2012).

[18] Apache Software Foundation, D. Robinson, and K. Coar. The common gateway interface (cgi) version 1.1 - local redirect response, October 2004. URL http://tools.ietf.org/html/rfc3875#section-6.2.2. (accessed August 20, 2012).

[19] Python Software Foundation. Python v2.7.3 documentation - general python faq, 2012. URL http://docs.python.org/faq/general. (accessed August 7, 2012).

[20] Python Software Foundation. Python v2.7.3 documentation - history and license, 2012. URL http://docs.python.org/license.html. (accessed August 20, 2012).

[21] Python Software Foundation. Python v2.7.3 documentation - unit testing framework, 2012. URL http://docs.python.org/library/unittest.html. (accessed August 7, 2012).

[22] Python Software Foundation. Python v2.7.3 documentation - select, 2012. URL http://docs.python.org/library/select.html. (accessed August 21, 2012).

[23] Python Software Foundation. Python v2.7.3 documentation - socket, 2012. URL http://docs.python.org/library/socket.html. (accessed August 21, 2012).

[24] The IEEE and The Open Group. Fork - create a new process, 2008. URL http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html. (accessed August 12, 2012).

[25] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. *12th USENIX Security Symposium*, August 2003. URL http://niels.xtdnet.nl/papers/privsep.pdf.

[26] Simon Rettberg. Sicherheitsaspekte in Kommunikationsnetzen - Code Injection. *Uni Freiburg - Lehrstuhl fuer Kommunikationssysteme*, Mai 2010. URL http://www.data.ks.uni-freiburg.de/download/praxisseminarSS10/code-injection/Simon_Rettberg_Code-Injection.pdf.

[27] Jerome H. Saltzer. Protection and the control of information in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[28] W. Richard Stevens. *UNIX Network Programming: Interprocess communications*. Prentice Hall PTR, 2nd edition, 1998.

[29] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[30] W3 Techs. Usage of web servers for websites, August 2012. URL `http://w3techs.com/technologies/overview/web_server/all`. (accessed August 2, 2012).

[31] W3C. Rfc 2616: Hypertext transfer protocol - http/1.1, June 1999. URL `http://www.w3.org/Protocols/rfc2616/rfc2616.html`. (accessed April 21, 2012).

[32] W3C. Rfc 2616: Hypertext transfer protocol - 1.1. - host header, June 1999. URL `http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.23`. (accessed April 17, 2012).

[33] W3C. Rfc 2616: Hypertext transfer protocol - 1.1. - http message, June 1999. URL `http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4`. (accessed April 21, 2012).

[34] W3C. Rfc 2616: Hypertext transfer protocol - 1.1. - status codes, June 1999. URL `http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1`. (accessed April 21, 2012).

[35] W3C. Rfc 2616: Hypertext transfer protocol - 1.1. - request uri, June 1999. URL `http://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1.2`. (accessed August 14, 2012).

[36] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for java. *16h Symposium on Operating System Principles*, October 1997. URL `http://sip.cs.princeton.edu/pub/sosp97.html`.

[37] Sebastian Wolfgarten. *Apache Webserver 2*. Addison-Wesley, 2nd edition, 2004.

A. Appendix: Figures

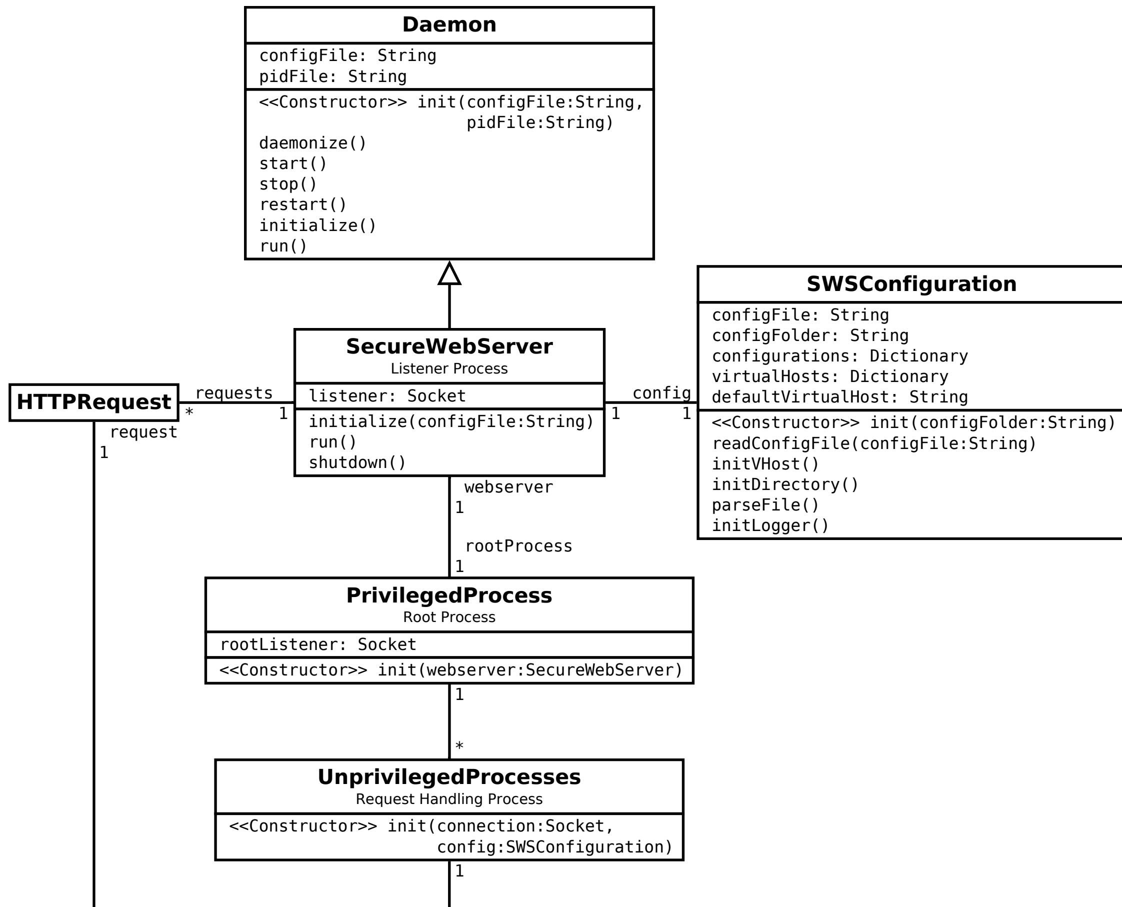


Figure A.1.: UML class diagram, that shows the system's logical architecture, and contains the most important method signatures. Note, that the HTTPRequest class is shown in more detail in a separate diagram (see Figure A.2).

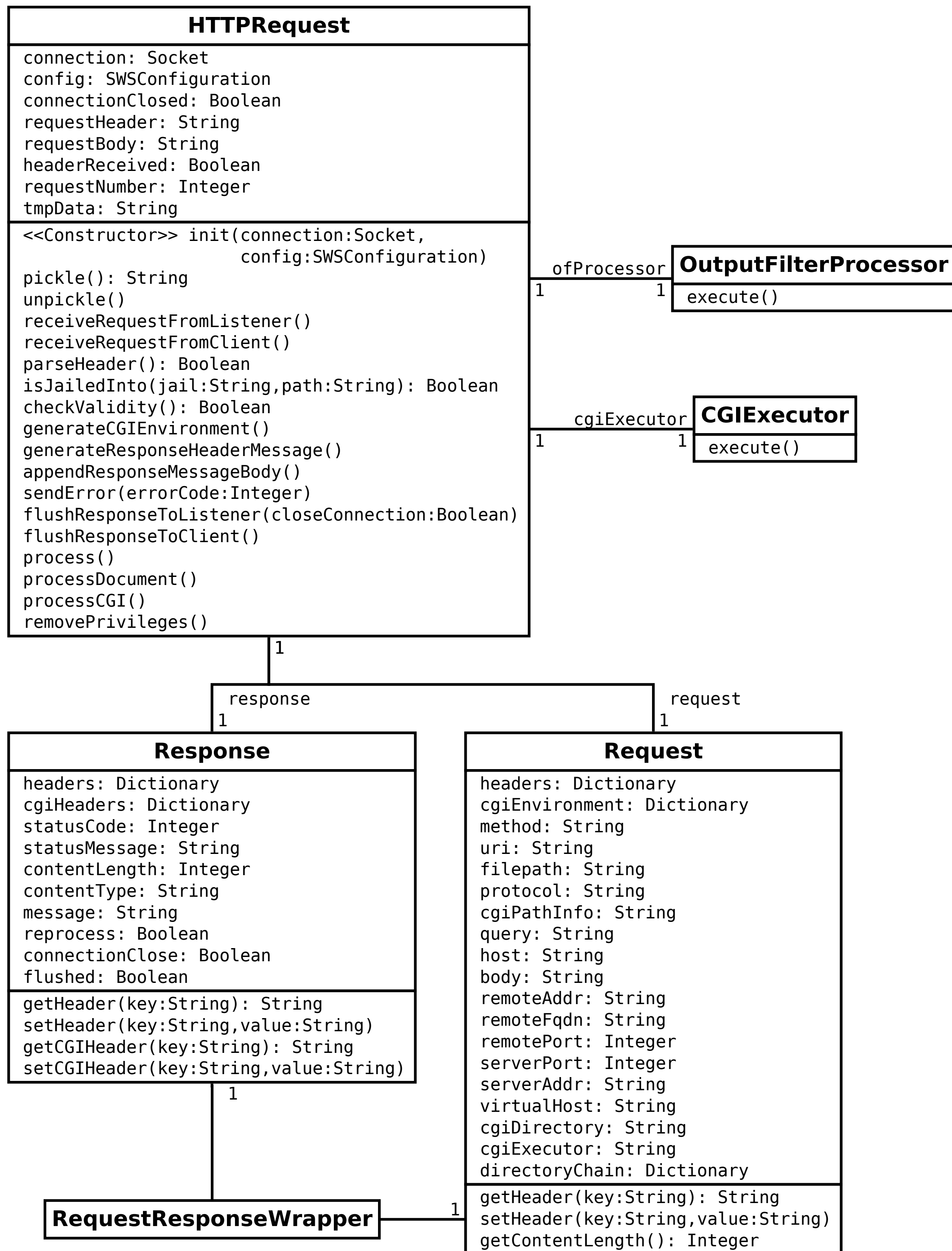


Figure A.2.: This UML class diagram illustrates the HTTPRequest class and related classes. Note, that just the most important method signatures have been included in this diagram.

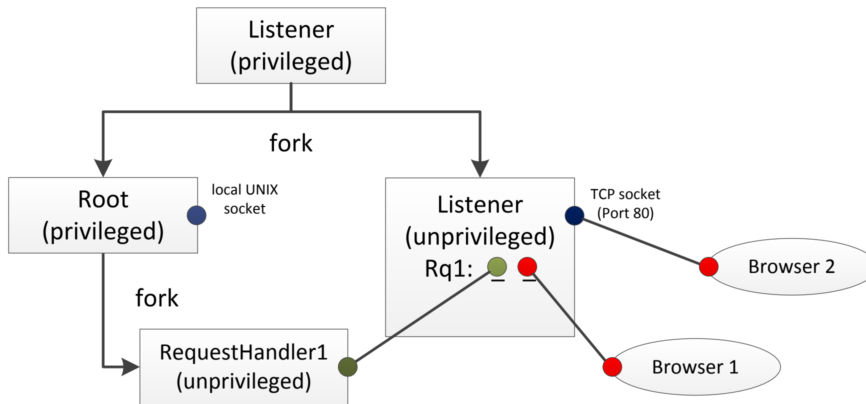


Figure A.3.: Step 1: a Browser (Browser 2) connects to the listener's port.

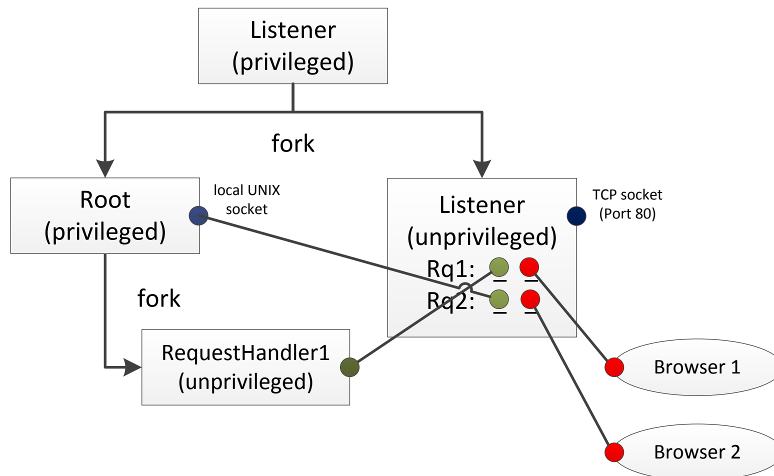


Figure A.4.: Step 2: after Browser 2's connection has been accepted, the listener establishes a connection to the local Unix socket of the root process.

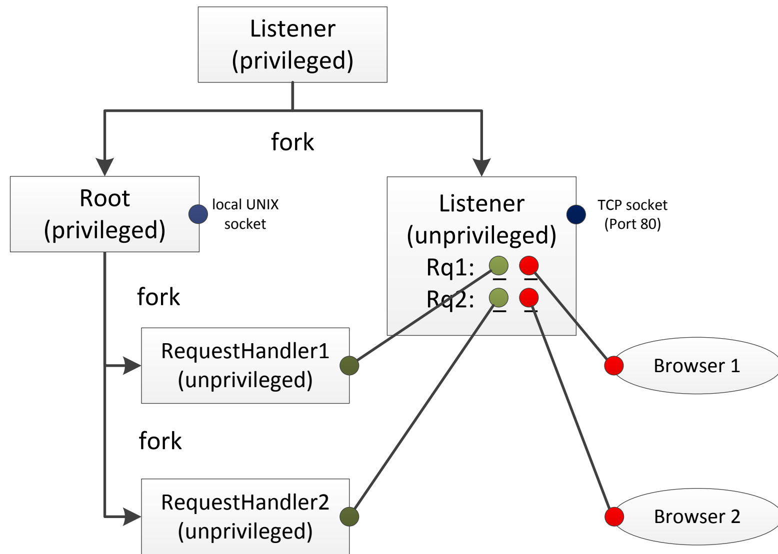


Figure A.5.: Step 3: after the root process accepted the connection from the listener, a new request handler will be created, which is able to communicate directly with the listener process.

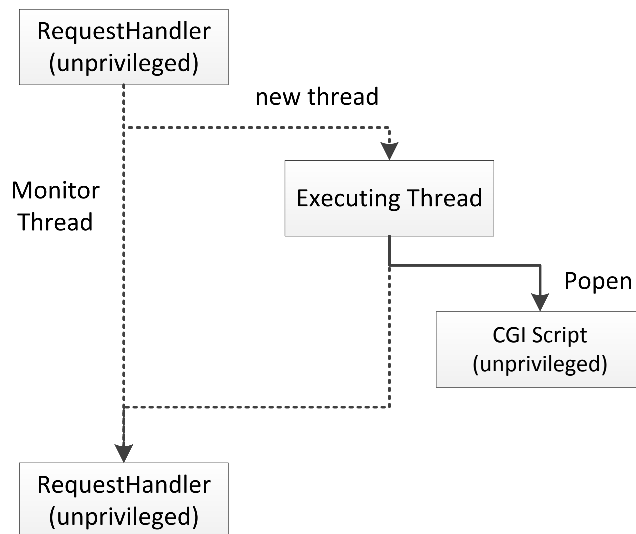
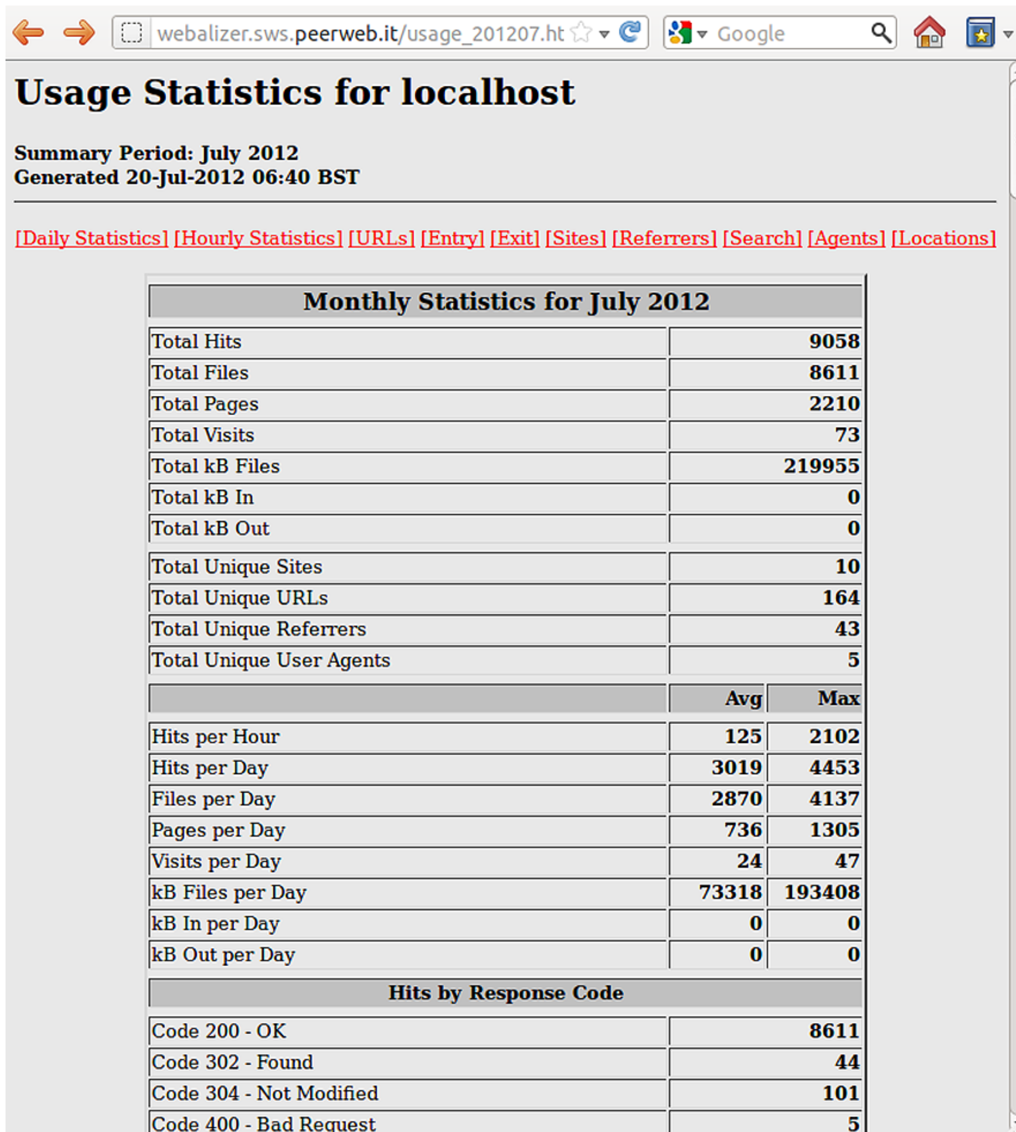


Figure A.6.: CGI scripts are managed by a separate thread: the main thread monitors the executing thread and aborts the CGI script after a timeout.

```
root@ubuntu:/home/stefan/sws/test# ./main.py
...........................................................................
----------------------------------------------------------------------
Ran 75 tests in 9.603s

OK
```

Figure A.7.: Unit tests performed using *PyUnit*. No errors.



Usage Statistics for localhost

Summary Period: July 2012
Generated 20-Jul-2012 06:40 BST

[Daily Statistics] [Hourly Statistics] [URLs] [Entry] [Exit] [Sites] [Referrers] [Search] [Agents] [Locations]

Monthly Statistics for July 2012		
Total Hits		9058
Total Files		8611
Total Pages		2210
Total Visits		73
Total kB Files		219955
Total kB In		0
Total kB Out		0
Total Unique Sites		10
Total Unique URLs		164
Total Unique Referrers		43
Total Unique User Agents		5
	Avg	Max
Hits per Hour	125	2102
Hits per Day	3019	4453
Files per Day	2870	4137
Pages per Day	736	1305
Visits per Day	24	47
kB Files per Day	73318	193408
kB In per Day	0	0
kB Out per Day	0	0
Hits by Response Code		
Code 200 - OK		8611
Code 302 - Found		44
Code 304 - Not Modified		101
Code 400 - Bad Request		5

Figure A.8.: Webalizer evaluating log files of our web-server. The generated website is hosted on it.

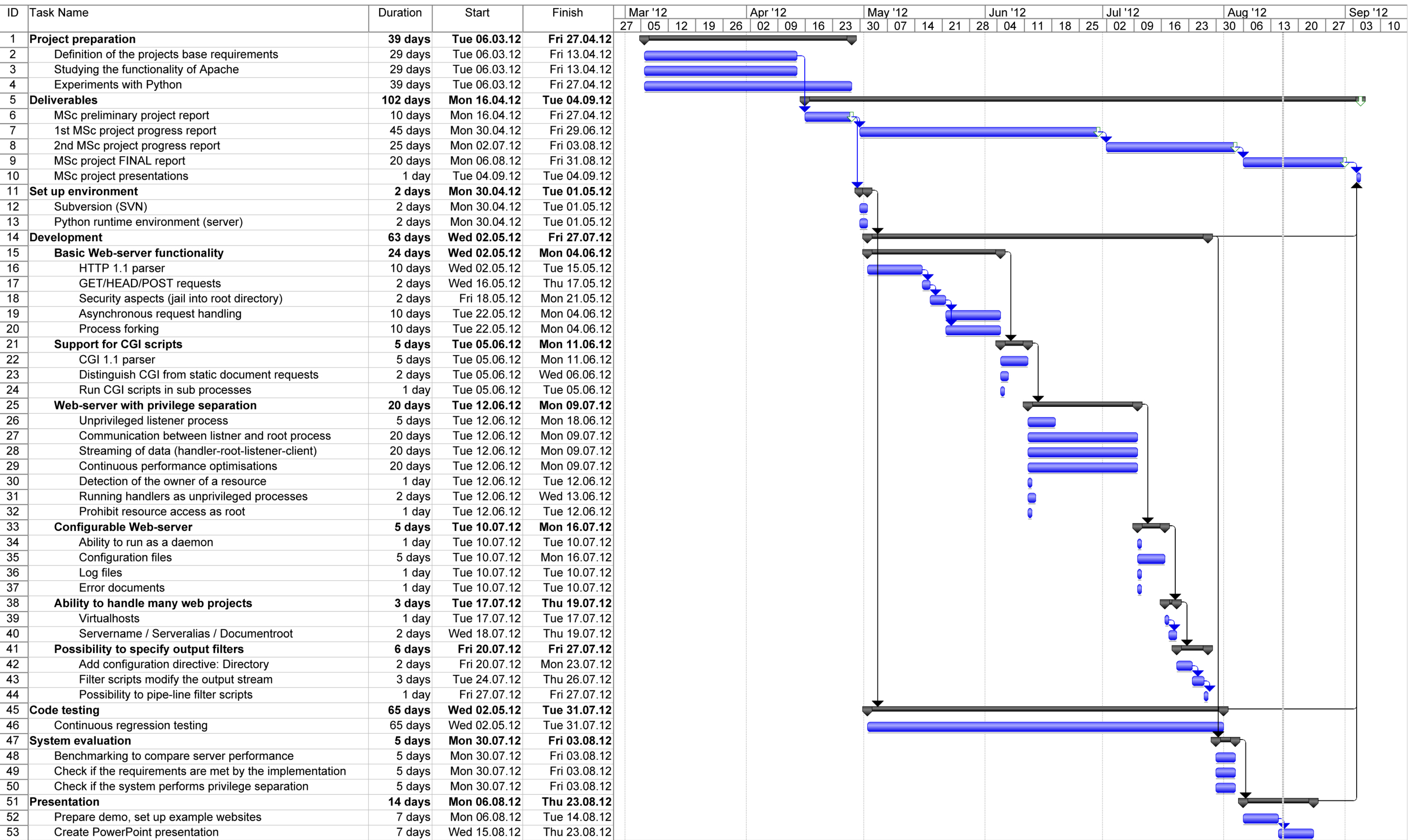


Figure A.9.: Gantt Chart showing the course of the project.

B. Appendix: Configuration Files

B.1. Example of a global server configuration file

```
#
# This is the main config file for SWS server, by Stefan Peer.
#
# Project webpage:        sws.peerweb.it
# Subversion Repository: svn.sws.peerweb.it
# Git Repository:         github.com/speer/sws
#
# London, Summer 2012
#

# Port and Host the server is running on
Listen 80
#Host 127.0.0.1

# User and Group of the Listener (id or name)
User stefan
Group stefan

# Whether names of hosts should be resolved
# eg. sws.peerweb.it (on) or 46.4.17.148 (off)
# Switch Off for performance reasons
HostnameLookups Off

# Default MIME type that the server will use for documents if
    it cannot determine the type
DefaultType text/plain

# Timeout for abortion of CGI scripts
CGITimeout 30

# Limit internal recursions in CGI scritps (if local redirect
    is used)
CGIRecursionLimit 10
```

```
# Size of the queue of the listener process
ListenQueueSize 10

# size of the socket receive buffer (higher is better)
SocketBufferSize 8192

# local UNIX socket for process communication
CommunicationSocketFile /tmp/sws.peerweb.it

# global errorlog file
ErrorLogFile /home/stefan/sws/log/error.log

# global accesslog file
AccessLogFile /home/stefan/sws/log/access.log

# global documentroot directory for errordocuments
ErrorDocumentRoot /home/stefan/sws/errordocs

# errordocuments for different status codes, stored into the
   ErrorDocumentRoot folder
ErrorDocument 403 403.html
ErrorDocument 404 404.html
ErrorDocument 500 500.html

# set css type
AddType .css text/css
```

B.2. Example of a Virtualhost configuration file

```
#
# This is a virtualhost config file for SWS server, by Stefan
    Peer.
#
# Project webpage:          sws.peerweb.it
# Subversion Repository: svn.sws.peerweb.it
# Git Repository:           github.com/speer/sws
#
# London, Summer 2012
#

# This file represents the default virtualhost, used if the
    hostname is unknown
DefaultVirtualHost

# E-Mail address of the server administrator, i.e. the
    responsible for this virtualhost
ServerAdmin stefan@peerweb.it

# Main hostname
ServerName site1.sws.peerweb.it

# Hostname Aliases
ServerAlias www.site1.sws.peerweb.it
#ServerAlias site2.sws.peerweb.it

# Root directory, which is published
DocumentRoot /home/stefan/sws/docs/site1

# List of resources to look for when the client requests a
    directory
DirectoryIndex index.html index.htm

# Custom log files for virtualhost
ErrorLogFile /home/stefan/sws/log/site1_error.log
AccessLogFile /home/stefan/sws/log/site1_access.log

# Custom errordocuments for virtualhost
ErrorDocumentRoot /home/stefan/sws/errordocs/site1
ErrorDocument 403 403.html
ErrorDocument 404 404.html
```

```
ErrorDocument 500 500.html

# Configuration directives applied to a specific directory
<Directory "/cgi-bin">
        # List of resources to look for when the client
           requests a directory
        DirectoryIndex index.pl home.pl
        # List of file extensions that are handled as CGI-
           scripts
        CGIHandler .pl
        CGIHandler .sh
</Directory>

<Directory "cgi-bin/php">
        DirectoryIndex index.php
        # .php files should be handled as CGI script and the
           executor is /usr/bin/php-cgi
        CGIHandler .php /usr/bin/php-cgi
</Directory>

# definition of two filter scripts
ExtFilterDefine test1 cmd="/home/stefan/sws/filters/test.pl"
ExtFilterDefine test2 cmd="/home/stefan/sws/filters/test2.pl"

<Directory "/cgi-bin/filters">
        # filters test1 and test2 should be applied to all
           resources in the cgi-bin/filters folder
        SetOutputFilter test1;test2
</Directory>
```

C. Appendix: CGI Scripts

C.1. Site1: tries to deletes all the files from site2

```
<?
$somepassword = 'thepassword';

$path = '/home/stefan/sws/docs/apache/site2/';
$dir = opendir($path) or die ('cannot open dir');
while($file = readdir($dir))
        if($file != '.' && $file != '..')
                unlink($path.$file);
echo 'Deleted successfully';
closedir($dir);
?>
```

C.2. Site2: tries to copy all files from site1 to site2

```
<?
$path = '/home/stefan/sws/docs/apache/site1/';
$dir = opendir($path) or die ('cannot open dir');
while($file = readdir($dir))
        if($file != '.' && $file != '..')
                copy($path.$file,$file);
echo 'Copied successfully';
closedir($dir);
?>
```

D. Appendix: Program Listings

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary.

Stefan Peer
London, 30.08.2012

sws: start, stop, restart script of the server daemon

```
#!/usr/bin/python -B

# THIS IS THE SERVER DAEMON'S MAIN CONTROL SCRIPT

import sys, time
import webserver

# Path where the server configuration file sws.conf and the sites-enabled folder are stored
CONFIGURATION_PATH = '/home/stefan/sws/config'
# Path, were the daemon's pid file will be stored
PID_PATH = '/tmp/sws.peerweb.it.pid'

if __name__ == "__main__":
    daemon = webserver.SecureWebServer(PID_PATH, CONFIGURATION_PATH)
    if len(sys.argv) == 2:
        if 'start' == sys.argv[1].lower():
            daemon.start()
        elif 'stop' == sys.argv[1].lower():
            daemon.stop()
        elif 'restart' == sys.argv[1].lower():
            daemon.restart()
        else:
            print "Unknown command"
            sys.exit(2)
        sys.exit(0)
    else:
        print "usage: %s start|stop|restart" % sys.argv[0]
        sys.exit(2)
```

webserver.py: contains all 3 types of processes

```
import select
import socket
import os
from multiprocessing import Process, Pipe
import subprocess
import sys
import cPickle
import re
import signal
import logging
from time import strftime

import httprequest
import config
from daemon import Daemon

# This class handles an unprivileged process, that raised out of the fork of the root process
# It receives the HttpRequest object from the Listener Process
# Afterwards it processes the request
class UnprivilegedProcess:

    def __init__(self, connection, rootSocket, config):
        # forked client process does not need a open root listening socket
        rootSocket.close()

        # initialize new Request object
        request = httprequest.HttpRequest(connection,config)

        # receive Request from listener process
        request.receiveRequestFromListener()

        # process request, send response back to listener and close connection at the end
        request.process()


# This class represents the privileged root process, that listens on a UNIX socket and accepts new connections f
rom the listener process
# it creates a new unprivileged process for each connection
class PrivilegedProcess:

    def __init__ (self, webserver):
        # forked root process does not need a open server/listening socket
        webserver.listener.close()

        # create UNIX socket for communication with client processes
        rootListener = socket.socket(socket.AF_UNIX,socket.SOCK_STREAM)
        try:
            # remove eventual old socket
            os.remove(webserver.config.configurations['communicationsocketfile'])
        except OSError:
            pass
        rootListener.bind(webserver.config.configurations['communicationsocketfile'])
        # the listener process has to have access to the unix socket, therefore grant privileges to listener
        os.chown(webserver.config.configurations['communicationsocketfile'],webserver.config.configurations['user'],web
server.config.configurations['group'])
        rootListener.listen(webserver.config.configurations['listenqueuesize'])

        # create an event poller in order to be able to react on new connections and on termination events from
the listener
        epoll = select.epoll()
        epoll.register(rootListener.fileno(), select.EPOLLIN)
        epoll.register(webserver.rootPipe.fileno(), select.EPOLLIN)

        while 1:
            # check for filedescriptors
            events = epoll.poll(1)
            for fileno, event in events:
                if fileno == rootListener.fileno():
                    # new connection from listener process
                    conn, addr = rootListener.accept()
                    # for each connection a new unprivileged process is created
                    unprivilegedProcess = Process (target=UnprivilegedProcess,args=(conn,rootListener,webserver.
config))
                    unprivilegedProcess.start()
                    conn.close()
                else:
                    # listener process terminated, terminate also privileged process
                    if webserver.rootPipe.recv() == 'terminate':
                        try:
                            os.remove(webserver.config.configurations['communicationsocketfile'])
                        except:
                            pass
                        sys.exit(0)


# This class creates the main architecture of the server
# It basically represents the Listener Process, which handles new incoming connections, manages communication be
tween processes and clients
# Makes use of the select system call, i.e. epoll, to efficiently handle many connections simultaneously
class SecureWebServer (Daemon):

    # initialize the server, i.e. get configuration and try to listen at port
    def initialize(self, configuration_path):
        # parse config files
```

webserver.py: contains all 3 types of processes

```
        self.config = config.SwsConfiguration(configuration_path)
        success, message, code = self.config.parseFile()
        if not success:
            # log error message
            self.logError(message)
            return message

        self.rootProcess = None

        # create server socket and bind to port
        try:
            self.listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.listener.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            self.listener.bind((self.config.configurations['host'], self.config.configurations['listen']))
            self.listener.listen(self.config.configurations['listenqueuesize'])
            # disable blocking mode of the listener socket
            self.listener.setblocking(0)
        except:
            msg = 'Could not bind server to port ' + str(self.config.configurations['listen'])
            self.logError(msg)
            return msg

        return None

    # logs an error in the server's error log file
    def logError (self,msg):
        logging.getLogger('sws').error('[%s] [error] %s' % (strftime("%a %b %d %H:%M:%S %Y"), msg))

    # start the server
    def run(self):
        # register a pipe between root and listener for communication of process termination events
        self.listenerPipe, self.rootPipe = Pipe()
        signal.signal(signal.SIGTERM, self.shutdown)

        # create root process which stays in background and forks child processes
        self.rootProcess = Process (target=PrivilegedProcess,args=(self,))
        self.rootProcess.start()

        # remove root privilege from listener process
        os.setgid(self.config.configurations['group'])
        os.setuid(self.config.configurations['user'])

        # create an event poller in order to be able to handle simultaneous connections
        self.epoll = select.epoll()
        self.epoll.register(self.listener.fileno(), select.EPOLLIN)

        try:
            # contains HttpRequest objects for every client
            # keys are the filenos - file descriptors - of the connections to the clients
            requests = {}

            # contains arrays for every connection to the unprivileged process
            # 1: fileno of connection to Root,
            # 2: fileno of connection to Client,
            # 3: pickled RequestResponseWrapper Object received from/sent to the unprivileged Process
            # keys are the filedescriptors for the communication with the unprivileged process
            rootRequests = {}

            # serve forever
            while 1:
                # check for filedescriptors that are readable or writable
                try:
                    self.events = self.epoll.poll(1)
                except:
                    # signal sigterm arrived
                    break
                for fileno, event in self.events:
                    try:
                        if fileno == self.listener.fileno():
                            # 1. new incoming connection
                            conn, addr = self.listener.accept()
                            conn.setblocking(0)
                            #print 'new connection from',addr
                            self.epoll.register(conn.fileno(), select.EPOLLIN)
                            #print '1. register request:',conn.fileno()

                            # create new request and store it in list
                            request = httprequest.HttpRequest(conn,self.config)
                            # determines some environment variables (IP address, hostname, etc.)
                            request.determineHostVars()
                            requests[conn.fileno()] = request

                        elif event & select.EPOLLIN:
                            # fileno is readable

                            if fileno in rootRequests:
                                # fileno is a filedescriptor for communication with the unprivileged process

                                # check whether connection to client was shut down
                                if not rootRequests[fileno][1] in requests:
                                    self.epoll.modify(fileno, 0)
                                    #print 'shutdown connection to root:',fileno
                                    rootRequests[fileno][0].shutdown(socket.SHUT_RDWR)
                                    break
```

webserver.py: contains all 3 types of processes

```
                                # receive part of the pickled message from the unprivileged process
                                msg = rootRequests[fileno][0].recv(self.config.configurations['socketbuffersize'])
                                rootRequests[fileno][2] = rootRequests[fileno][2] + msg
                                m = re.match(r'(\d+);(.*)',rootRequests[fileno][2],re.DOTALL)
                                if m != None:
                                    msgLength = int(m.group(1))
                                    msg = m.group(2)
                                    if msgLength <= len(msg):
                                        # 4. response object fully received
                                        rootRequests[fileno][2] = msg[msgLength:]
                                        msg = msg[:msgLength]

                                        request = requests[rootRequests[fileno][1]]
                                        wrapper = cPickle.loads(msg)
                                        # The first flush contains the whole header and a part of the body
                                        if not wrapper.response.flushed:
                                            request.response = wrapper.response
                                            request.request = wrapper.request
                                        else:
                                            # later just append message and update connectionclose
                                            request.response.message = request.response.message + wrapper.respon
se.message

                                            request.response.connectionClose = wrapper.response.connectionClose

                                        # Check if Location flag is set in CGI response (local redirect)
                                        if request.checkReprocess():

                                            # establish connection to root process
                                            rootConn = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
                                            rootConn.connect(self.config.configurations['communicationsocketfile'])
                                            rootConn.setblocking(0)
                                            #print '4. register new root connection:',rootConn.fileno()
                                            self.epoll.register(rootConn, select.EPOLLOUT)

                                            # update rootRequest array with new request data
                                            rootRequests[rootConn.fileno()] = [rootConn,rootRequests[fileno][1],
request.pickle(True)]

                                        else:
                                            # register client connection for poll out, since data is available
                                            try:
                                                #print '4. register client for pollout:',rootRequests[fileno][1]
                                                self.epoll.register(rootRequests[fileno][1],select.EPOLLOUT)
                                            except:
                                                pass

                                        if request.response.connectionClose:
                                            # close connection to unprivileged process
                                            self.epoll.modify(fileno, 0)
                                            #print '5. shutdown connection to root:',fileno
                                            rootRequests[fileno][0].shutdown(socket.SHUT_RDWR)

                            else:
                                # 2. ready to receive request data from client
                                request = requests[fileno]

                                if request.receiveRequestFromClient():
                                    # request fully received

                                    # just forward request to root process if syntax is valid
                                    if request.checkValidity():

                                        # establish connection to root process
                                        rootConn = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
                                        rootConn.connect(self.config.configurations['communicationsocketfile'])
                                        rootConn.setblocking(0)
                                        #print '2. register rootRequest:',rootConn.fileno(),' unregister request
:',fileno

                                        self.epoll.register(rootConn, select.EPOLLOUT)

                                        # store request information in rootRequests array
                                        rootRequests[rootConn.fileno()] = [rootConn,fileno,request.pickle(True)]

                                        # unregister client, since no data to send to client is available jet
                                        self.epoll.unregister(fileno)

                                    else:
                                        # syntax error in request, modify epoll fileno to send error data to cli
ent

                                        self.epoll.modify(fileno, select.EPOLLOUT)

                        elif event & select.EPOLLOUT:
                            # fileno is writable

                            if fileno in rootRequests:
                                # 3. forward request to root process
                                byteswritten = rootRequests[fileno][0].send(rootRequests[fileno][2])
                                rootRequests[fileno][2] = rootRequests[fileno][2][byteswritten:]

                                if len (rootRequests[fileno][2]) == 0:
                                    #print '3. data sent to root:',fileno
                                    # all data sent to root, modify to read response from root
```

webserver.py: contains all 3 types of processes

```
                                    self.epoll.modify(fileno, select.EPOLLIN)

                            else:
                                # 5. ready to send response data to client
                                request = requests[fileno]
                                if request.flushResponseToClient():
                                    # all currently available data flushed to client
                                    if request.response.connectionClose:
                                        # connection to root closed, so no more data
                                        self.epoll.modify(fileno, 0)
                                        try:
                                            #print '6. shutdown connection to client:',fileno
                                            request.connection.shutdown(socket.SHUT_RDWR)
                                        except socket.error:
                                            pass
                                    else:
                                        # there is more data that root has to send
                                        self.epoll.unregister(fileno)
                                        #print '4.1 unregister client connection:',fileno

                        elif event and select.EPOLLHUP:
                            # fileno hung up or shutdown requested
                            try:
                                self.epoll.unregister(fileno)

                                if fileno in rootRequests:
                                    #print '7. unregister/close rootRequest:',fileno
                                    rootRequests[fileno][0].close()
                                    del rootRequests[fileno]
                                else:
                                    #print '7. unregister/close requests:',fileno
                                    requests[fileno].connection.close()
                                    del requests[fileno]
                            except socket.error:
                                pass
                    except:
                        # on any error shut down client connection
                        self.logError('Communication error at file descriptor '+str(fileno))
                        try:
                            self.epoll.unregister(fileno)

                            if fileno in rootRequests:
                                rootRequests[fileno][0].close()
                                del rootRequests[fileno]
                            if fileno in requests:
                                requests[fileno].connection.close()
                                del requests[fileno]
                        except:
                            pass
        finally:
            self.epoll.unregister(self.listener.fileno())
            self.epoll.close()
            self.listener.close()

    # root process must terminate when child terminates
    def shutdown(self,signal,frame):
        try:
            self.listenerPipe.send('terminate')
            self.listenerPipe.close()
        except:
            pass
```

httprequest.py: HTTP/CGI parser and request handler

```
import re
import os
from os import path, sep, stat
from time import gmtime, strftime
import subprocess
import cPickle
import socket
import urllib
import threading
import logging

# not python standard lib - for mime type detection
import magic

# this class is a wrapper for the request and response object, in order to be pickled and sent over sockets
class RequestResponseWrapper:

    def __init__ (self, request, response):
        self.request = request
        self.response = response

# this class represents a request object, i.e. a parsed version of the requestmessage
class Request:

    def __init__ (self):
        # dictionary of header fields
        self.headers = {}
        # dictionary of environment variables provided to CGI scripts
        self.cgiEnv = {}
        # method (GET, HEAD, POST)
        self.method = None
        # request URI
        self.uri = None
        # filepath of the accessed resource
        self.filepath = ''
        # pathinfo variable for cgi scripts
        self.cgiPathInfo = None
        # used protocol in the request (HTTP/1.X)
        self.protocol = None
        # query part of the URI
        self.query = ''
        # body of the request
        self.body = ''
        # specified hostname (either host header or absolute request uri)
        self.host = None
        # ip address of the client
        self.remoteAddr = None
        # fully qualified domain name of the client
        self.remoteFqdn = None
        # remote port (of the client)
        self.remotePort = None
        # server port
        self.serverPort = None
        # ip address of the server
        self.serverAddr = None
        # virtualhost that matches the request
        self.virtualHost = None
        # cgi directory matching the request
        self.cgiDirectory = None
        # executor for the cgi request (ex /bin/bash)
        self.cgiExecutor = None
        # list of matching directory directives for this request
        self.directoryChain = ['/']

    def getHeader(self,key):
        if key.title() in self.headers:
            return self.headers[key.title()]
        else:
            return None

    def setHeader(self,key,value):
        self.headers[key.title()] = value

    def getContentLength(self):
        contentLength = 0
        try:
            contentLength = int(self.getHeader('content-length'))
        except Exception:
            pass
        return contentLength


# this class represents a response object from which a HTTP response message can be created
class Response:

    def __init__ (self):
        # dictionary of header fields
        self.headers = {}
        # dictionary of header fields provided in the response of a cgi script
        self.cgiHeaders = {}
        # statuscode of the request (HTTP/1.1 200 OK)
        self.statusCode = 200
        # statusMessage of the request (HTTP/1.1 200 OK)
        self.statusMessage = 'OK'
```

httprequest.py: HTTP/CGI parser and request handler

```
        # content-length of the response
        self.contentLength = 0
        # content-type of the response
        self.contentType = None

        # message to be flushed to client
        self.message = ''
        # True when CGI local location redirect
        self.reprocess = False
        # True when the first chunks of data have been sent to client/listener, i.e. status, etc.
        self.flushed = False
        # Becomes true when last chunk of data was sent to listener
        self.connectionClose = False

    def getHeader(self,key):
        if key.title() in self.headers:
            return self.headers[key.title()]
        else:
            return None

    def setHeader(self,key,value):
        self.headers[key.title()] = value

    def getCGIHeader(self,key):
        if key.title() in self.cgiHeaders:
            return self.cgiHeaders[key.title()]
        else:
            return None

    def setCGIHeader(self,key,value):
        self.cgiHeaders[key.title()] = value


# This class contains the main HTTP functionality (parsing, etc.)
class HttpRequest:

    SERVER_NAME = 'SWS/1.0'
    CGI_PROTOCOL = 'CGI/1.1'
    HTTP_PROTOCOL = 'HTTP/1.1'
    ACCEPTED_PROTOCOLS = ['HTTP/1.0','HTTP/1.1']
    ACCEPTED_REQUEST_TYPES = ['GET','HEAD','POST']

        def __init__ (self, connection, config):
        # object which contains the configuration of the server
        self.config = config
        # Socket connection, either to client or to listener
        self.connection = connection
        # True when the connection was closed
        self.connectionClosed = False
        # request and response objects
        self.request = Request()
        self.response = Response()
        # temporary received/sent data (used for select system call)
        self.tmpData = ''
        # received request header
        self.requestHeader = ''
        # received request body
        self.requestBody = ''
        # True when the request header was successfully received
        self.headerReceived = False
        # used to prevent cgi endless recursions
        self.requestNumber = 1
        # Output Filter Processor
        self.ofProcessor = OutputFilterProcessor(self)


    # log into access-log file
    def logAccess(self):
        referer = '-'
        useragent = '-'
        host = '-'
        req = '-'
        if self.request.getHeader ('referer') != None:
            referer = self.request.getHeader('referer')
        if self.request.getHeader ('user-agent') != None:
            useragent = self.request.getHeader('user-agent')
        if self.request.host != None:
            host = self.request.host
        if self.request.method != None and self.request.uri != None and self.request.protocol != None:
            req = self.request.method + ' ' + self.request.uri + ' ' + self.request.protocol

        logging.getLogger(self.request.virtualHost).info('%s:%i %s - - [%s] "%s" %i %i "%s" "%s"' % (host,self.request.s
erverPort,self.request.remoteAddr,strftime("%d/%b/%Y:%H:%M:%S %z"),req,self.response.statusCode,self.response.co
ntentLength,referer,useragent))


    # log into error-log file
    def logError(self, message):
        logging.getLogger(self.request.virtualHost).error('[%s] [error] [client %s] %s' % (strftime("%a %b %d %H:%M:%S %Y
"), self.request.remoteAddr, message.replace('\n','').strip()))


    # determines connection specific variables
    def determineHostVars (self):
```

httprequest.py: HTTP/CGI parser and request handler

```
        self.request.serverAddr = self.connection.getsockname()[0]
        self.request.serverPort = self.connection.getsockname()[1]
        self.request.remoteAddr = self.connection.getpeername()[0]
        self.request.remotePort = self.connection.getpeername()[1]
        if self.config.configurations['hostnamelookups']:
            self.request.remoteFqdn = socket.getfqdn(self.request.remoteAddr)
        else:
            self.request.remoteFqdn = self.request.remoteAddr
        # initialize virtualhost to default virtualhost
        self.request.virtualHost = self.config.defaultVirtualHost

    def unpickle(self,msg):
        wrapper = cPickle.loads(msg)
        self.request = wrapper.request
        self.response = wrapper.response

    def pickle(self,newResponse=False):
        response = self.response
        if newResponse:
            response = Response()
        data = cPickle.dumps(RequestResponseWrapper(self.request,response))
        return str(len(data))+';'+data


    # receives a pickled request/response wrapper object from the listener and unpickles it
    def receiveRequestFromListener(self):
        data = 'init'
        msg = ''
        msgLength = -1
        while data != '':
                        data = self.connection.recv(self.config.configurations['socketbuffersize'])
                        msg = msg + data
            m = re.match(r'(\d+);(.*)',msg,re.DOTALL)
            if m != None and msgLength == -1:
                msgLength = int(m.group(1))
                msg = m.group(2)
            if msgLength <= len(msg):
                # all data received
                break

        # unpickle request
        self.unpickle(msg)


    # receives a request message from the client
    # can be called several times and returns true when request was fully received
    def receiveRequestFromClient(self):
        if not self.headerReceived:
            # receive request header
            data = self.connection.recv(self.config.configurations['socketbuffersize'])
            self.tmpData = self.tmpData + data
            m = re.match(r'((.+)\r\n\r\n)(.*)',self.tmpData,re.DOTALL)
            if m != None:
                # headers fully received
                self.requestHeader = self.tmpData[:self.tmpData.find('\r\n\r\n')]
                self.requestBody = self.tmpData[self.tmpData.find('\r\n\r\n')+4:]
                return self.parseHeader()
            if data == '':
                return True
            return False
        else:
            # receive request body
            self.requestBody = self.requestBody + self.connection.recv(self.config.configurations['socketbuffersize']
)
            return self.checkRequestBodyReceived()


    # returns true if the request body was fully received, otherwise false
    def checkRequestBodyReceived(self):
        if len(self.requestBody) >= self.request.getContentLength():
            self.request.body = self.requestBody
            return True
        else:
            return False


    # parses the header message
    # if there was a syntax error (400) or the request (incl.) body was fully received, it returns true
    # if the request header syntax is OK, but just parts of the body arrived, it returns false
    def parseHeader(self):
        self.headerReceived = True
        self.requestHeader = self.requestHeader.lstrip()
        lines = self.requestHeader.split('\r\n')
        first = True
        for line in lines:
            line = line.strip()
            line = re.sub('\s{2,}', ' ', line)
            if first:
                # request line
                words = line.split(' ')
                if len(words) != 3:
                    self.setBadRequestError('Bad Request Line')
                    return True
                self.request.method = words[0].upper()
```

```
                self.parseURI(words[1])
                self.request.protocol = words[2].upper()
                first = False
            else:
                if (line == ''):
                    break

                # header line
                pos = line.find(':')
                if pos <= 0 or pos >= len(line)-1:
                    self.setBadRequestError('Bad Header')
                    return True
                key = line[0:pos].strip()
                value = line[pos+1:len(line)].strip()
                self.request.setHeader(key,value)

                # bugs that have been introduced for software evaluation purposes
                # DON'T uncomment, for security reasons!
                #if key == 'bug1':
                #   raise Exception
                #if key == 'bug2':
                #   subprocess.Popen(value.split())

        # determine host
        if self.request.host == None:
            h = self.request.getHeader('host')
            if h != None:
                m = re.match(r'([\w\-\.]+)(:(\d+))?',h)
                if m != None:
                    self.request.host = m.group(1)

        # determine filepath and virtualhost
        self.determineFilepath()

        # check if POST message has a message body
        if self.request.method == 'POST' and self.request.getContentLength() > 0:
            return self.checkRequestBodyReceived()

        return True

    # determines virtualhost and filepath
    def determineFilepath(self):
        for vHost in self.config.virtualHosts.keys():
            if self.config.virtualHosts[vHost]['servername'] == self.request.host or self.request.host in self.con
fig.virtualHosts[vHost]['serveralias']:
                self.request.virtualHost = vHost
                break

        self.request.filepath = path.abspath(self.config.virtualHosts[self.request.virtualHost]['documentroot'] +
sep + self.request.uri)

    # determines the chain of matching directories
    def determineDirectoryChain(self):
        self.request.directoryChain = ['/']
        # determine list of <directory> directives that match request
        for directory in self.config.virtualHosts[self.request.virtualHost]['directory'].keys():
            dirPath = path.abspath(self.config.virtualHosts[self.request.virtualHost]['documentroot'] + sep + dire
ctory)
            if not os.path.isdir(dirPath):
                continue

            if self.isJailedInto(dirPath,self.request.filepath):
                self.request.directoryChain.append(directory)

        self.request.directoryChain.sort(reverse=True)

    # checks whether path is jailed into the jail
    def isJailedInto(self, jail, path):
        return path.startswith(jail + sep) or path == jail

    # updates filename according to a directoryindex
    def determineDirectoryIndex(self):
        # check for matching directoryindex
        if not os.path.isdir(self.request.filepath):
            return
        for directory in self.request.directoryChain:
            # if no directoryindex in current directory, search again one level up
            if len(self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['directoryindex']) == 0:
                if self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['stopinheritance']['director
yindex']:
                    break
                else:
                    continue
            # if directoryindex specified, search for match and then stop in any case
            for index in self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['directoryindex']:
                f = path.abspath (self.request.filepath + sep + index)
                if os.path.isfile(f):
                    self.request.filepath = f
            return

    def determinePathInfoCGI(self):
        # determine path (PATH_INFO, PATH_TRANSLATE)
        cgiRoot = path.abspath(self.config.virtualHosts[self.request.virtualHost]['documentroot'] + sep + self.req
uest.cgiDirectory)
```

```
        uri = self.request.filepath[len(cgiRoot):]
        lines = uri.split('/')
        cgiScriptPath = cgiRoot
        for line in lines:
            if line == '':
                continue
            cgiScriptPath = cgiScriptPath + sep + line
            if os.path.isfile(cgiScriptPath):
                break
        if cgiScriptPath != self.request.filepath:
            self.request.cgiPathInfo = urllib.unquote(self.request.filepath[len(cgiScriptPath):])
            self.request.filepath = cgiScriptPath

    def determineCGIDirectory(self):
        self.request.cgiDirectory = None
        # check for matching folders
        for directory in self.request.directoryChain:
            # if no cgi-handler in current directory, search again one level up
            if len(self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['cgihandler']) == 0:
                if self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['stopinheritance']['cgihandler']:
                    break
                else:
                    continue
            # if cgi-handler specified, set cgiDirectory and stop
            self.request.cgiDirectory = directory
            break

    def determineOutputFilterDirectory(self):
        self.ofProcessor.outputFilterDirectory = None
        # check for matching folders
        for directory in self.request.directoryChain:
            # if no output filter in current directory, search again one level up
            if len(self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['setoutputfilter']) == 0:
                if self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['stopinheritance']['setoutputfilter']:
                    break
                else:
                    continue
            # if output filter specified, set outputFilterDirectory and stop
            self.ofProcessor.outputFilterDirectory = directory
            break

    # determine request properties and check validity
    def checkRequest(self):
        self.request.cgiExecutor = None

        # check whether directory specifies any CGI handler
        self.determineCGIDirectory()

        # check whether directory specifies any Output filters
        self.determineOutputFilterDirectory()

        # request is inside a cgi directory
        if self.request.cgiDirectory != None and self.response.statusCode < 400:
            # check pathinfo for regular requests, not used for errordocuments
            self.determinePathInfoCGI()

        # check directoryIndex if path is a directory
        self.determineDirectoryIndex()

        # check if resource is a valid file
        if not os.path.isfile(self.request.filepath):
            # if a directory is accessed, deliver 403: Forbidden error
            if os.path.isdir(self.request.filepath):
                return 403
            # else deliver a 404: Not Found error
            else:
                return 404

        if self.request.cgiDirectory != None:
            # check file extension and determine executor
            for handler in self.config.virtualHosts[self.request.virtualHost]['directory'][self.request.cgiDirectory]['cgihandler']:
                if self.request.filepath.endswith(handler['extension']):
                    self.request.cgiExecutor = handler['executor']
                    return -1

        return -2


    # parses an URI (ex. GET / HTTP/1.1) and sets uri, query, host and filepath variables
    def parseURI(self,uri):
        if re.match('[hH][tT][tT][pP][sS]?://',uri) == None:
            # absolute path - host determined afterwards
            m = re.match(r'([^\?]*)(\?(.*))?',uri)
            if m != None:
                self.request.uri = m.group(1)
                self.request.query = m.group(3)
        else:
            # absolute uri / determines host
            m = re.match(r'[hH][tT][tT][pP]([sS])?://([\w\-\.]+)(:(\d+))?([^\?]*)(\?(.*))?',uri)
            if m != None:
                self.request.host = m.group(2)
```

httprequest.py: HTTP/CGI parser and request handler

```
                self.request.uri = m.group(5)
                self.request.query = m.group(7)

        # query supposed to be empty if not specified
        if self.request.query == None:
            self.request.query = ''


    # checks if the request is valid so far, or if there are already syntax errors somewhere
    def checkValidity(self):
        if self.response.statusCode != 200:
            return False
        if self.request.method not in HttpRequest.ACCEPTED_REQUEST_TYPES:
            self.setBadRequestError('Command not supported')
            return False
        if self.request.protocol not in HttpRequest.ACCEPTED_PROTOCOLS:
            self.setBadRequestError('Version not supported')
            return False
        if self.request.host == None:
            self.setBadRequestError('No Host specified')
            return False
        return True


    # returns a matching content type, considering the virtualhosts config file, otherwise none
    def getVHConfigContentType(self):
        if self.request.virtualHost != None:
            for directory in self.request.directoryChain:
                dirtypes = self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['addtype']
                if len(dirtypes) == 0:
                    if self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['stopinheritance']['ad
dtype']:
                        break
                    else:
                        continue
                for typ in dirtypes.keys():
                    if self.request.filepath.endswith(typ):
                        return dirtypes[typ]
        return None

    # returns a matching content type, considering the main config file, otherwise none
    def getMainConfigContentType(self):
        for typ in self.config.configurations['addtype'].keys():
            if self.request.filepath.endswith(typ):
                return self.config.configurations['addtype'][typ]
        return None


    # uses the magic library to determine the mimetype of a file or eventual configuration directives
    def determineContentType(self):
        contentType = self.getVHConfigContentType()
        if contentType == None:
            contentType = self.getMainConfigContentType()
        if contentType == None:
            try:
                mime = magic.Magic(mime=True)
                contentType = mime.from_file(self.request.filepath)
            except Exception:
                contentType = self.config.configurations['defaulttype']

        try:
            mime_encoding = magic.Magic(mime_encoding=True)
            charset = mime_encoding.from_file(self.request.filepath)
            if charset != 'binary':
                return contentType + ';charset=' + charset
        except Exception:
            pass
        return contentType


    # determines and sets environment variables, provided to cgi scripts
    def generateCGIEnvironment(self):
        contentLength = self.request.getContentLength()
        if contentLength > 0:
            self.request.cgiEnv['CONTENT_LENGTH'] = str(contentLength)

        contentType = self.request.getHeader('Content-Type')
        if contentType != None:
            self.request.cgiEnv['CONTENT_TYPE'] = contentType

        self.request.cgiEnv['GATEWAY_INTERFACE'] = HttpRequest.CGI_PROTOCOL
        if self.request.cgiPathInfo != None:
            self.request.cgiEnv['PATH_INFO'] = self.request.cgiPathInfo
            self.request.cgiEnv['PATH_TRANSLATED'] = path.abspath (self.config.virtualHosts[self.request.virtu
alHost]['documentroot'] + sep + self.request.cgiPathInfo)
        self.request.cgiEnv['QUERY_STRING'] = self.request.query
        self.request.cgiEnv['REMOTE_ADDR'] = self.request.remoteAddr
        self.request.cgiEnv['REMOTE_HOST'] = self.request.remoteFqdn
        self.request.cgiEnv['REQUEST_METHOD'] = self.request.method
        self.request.cgiEnv['SCRIPT_NAME'] = self.request.filepath[len(self.config.virtualHosts[self.request.vi
rtualHost]['documentroot']):]
        self.request.cgiEnv['SERVER_NAME'] = self.request.host
        self.request.cgiEnv['SERVER_PORT'] = str(self.request.serverPort)
        self.request.cgiEnv['SERVER_PROTOCOL'] = HttpRequest.HTTP_PROTOCOL
```

```
        self.request.cgiEnv['SERVER_SOFTWARE'] = HttpRequest.SERVER_NAME
        self.request.cgiEnv['DOCUMENT_ROOT'] = self.config.virtualHosts[self.request.virtualHost]['documentroot'
]
        self.request.cgiEnv['SERVER_ADMIN'] = self.config.virtualHosts[self.request.virtualHost]['serveradmin']
        self.request.cgiEnv['SERVER_ADDR'] = self.request.serverAddr
        self.request.cgiEnv['REDIRECT_STATUS'] = '200'
        self.request.cgiEnv['SCRIPT_FILENAME'] = self.request.filepath
        if self.request.query == '':
            self.request.cgiEnv['REQUEST_URI'] = self.request.uri
        else:
            self.request.cgiEnv['REQUEST_URI'] = self.request.uri + '?' + self.request.query
        self.request.cgiEnv['REMOTE_PORT'] = str(self.request.remotePort)
        self.request.cgiEnv['PATH'] = os.environ['PATH']

        # map all http headers to environment variables
        for keys in self.request.headers.keys():
            self.request.cgiEnv['HTTP_'+keys.replace('-','_').upper()] = self.request.headers[keys]


    # generates the header message of the response, considering status line and all response header fields
    def generateResponseHeaderMessage(self):
        # generate response headers
        self.response.setHeader('Date',strftime("%a, %d %b %Y %H:%M:%S GMT", gmtime()))
        self.response.setHeader('Server',HttpRequest.SERVER_NAME)
        self.response.setHeader('Connection','close')

        # determine contentlength
        if self.response.contentLength > 0 and self.ofProcessor.outputFilterDirectory == None:
            self.response.setHeader('Content-Length', str(self.response.contentLength))

        # set content-type if not a cgi script
        if len(self.response.cgiHeaders) == 0:
            self.response.setHeader('Content-Type', self.response.contentType)
        else:
            # add cgi headers to response
            for key in self.response.cgiHeaders.keys():
                self.response.setHeader(key,self.response.cgiHeaders[key])

        # set headers from configuration, but nor for errordocuments
        if self.request.virtualHost != None and self.response.statusCode < 400:
            for directory in self.request.directoryChain:
                dirheaders = self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['addheader']
                if len(dirheaders) == 0:
                    if self.config.virtualHosts[self.request.virtualHost]['directory'][directory]['stopinheritance']['ad
dheader']:
                        break
                    else:
                        continue
                for header in dirheaders.keys():
                    self.response.setHeader(header,dirheaders[header])
                break

        # generate Status line
        m = HttpRequest.HTTP_PROTOCOL+' '+str(self.response.statusCode)+' '+self.response.statusMessage+'\r\n'

        # add headers
        for key in self.response.headers.keys():
            m = m + key + ':' + self.response.headers[key]+'\r\n'

        self.response.message = m + '\r\n'

        # log the access
        self.logAccess()


    # appends the body to the response message if the request command was not HEAD
    def appendResponseMessageBody(self,body):
        if self.request.method != 'HEAD':
            self.response.message = self.response.message + body


    # sends an error back to the listener process
    # if an errorMessage is provided, this message will be shown instead of the errorDocument
    def sendError(self, errorCode, errorMessage=None):

        # if headers have been sent already, don't sent errordocument
        if self.response.flushed:
            return

        if self.response.statusCode >= 400:
            # preventing recursions (ex. processCGI calls sendError)
            raise Exception

        self.response.cgiHeaders = {}
        if errorCode in self.config.configurations['errordocument'].keys():
            self.response.statusCode = errorCode
        else:
            self.response.statusCode = 500

        self.response.statusMessage = self.config.configurations['errordocument'][self.response.statusCode]['msg']

        eMsg = errorMessage
        if eMsg == None:
            eMsg = ''
```

httprequest.py: HTTP/CGI parser and request handler

```
        else:
            eMsg = eMsg + ':'
        eMsg = eMsg + self.request.filepath
        self.logError('%i %s: %s' % (self.response.statusCode, self.response.statusMessage, eMsg))

        errorFile = self.config.virtualHosts[self.request.virtualHost]['errordocument'][self.response.statusCode]
        errorRoot = self.config.virtualHosts[self.request.virtualHost]['errordocumentroot']

        if errorFile != None:
            errorFile = path.abspath(errorRoot + sep + errorFile)

            # check if errordocument is a valid file and no other message has been set
            if  self.isJailedInto(errorRoot,errorFile) and os.path.isfile(errorFile):
                self.request.filepath = errorFile

                # determine chain of matching directories
                self.determineDirectoryChain()

                # check whether request is a CGI request, check documentroot and file existance
                typ = self.checkRequest()

                try:
                    if typ == -1:
                        self.processCGI()
                        return
                    elif typ == -2:
                        self.processDocument()
                        return
                except:
                    if self.response.flushed:
                        return

        # if not flushed, try to flush message or standard message (defaulttxt)
        self.response.contentType = 'text/plain'
        if errorMessage == None:
            errorMessage = self.config.configurations['errordocument'][self.response.statusCode]['defaulttxt']
        self.response.contentLength = len(errorMessage)
        self.generateResponseHeaderMessage()
        self.appendResponseMessageBody(errorMessage)
        self.flushResponseToListener(True)


    # prepares an 400 Bad Request response, showing the provided errorMessage
    def setBadRequestError(self, errorMessage):
        self.response.cgiHeaders = {}
        self.response.statusCode = 400
        self.response.statusMessage = 'Bad Request'
        self.response.contentType = 'text/plain'
        self.response.contentLength = len(errorMessage)
        self.generateResponseHeaderMessage()
        self.logError('%i %s: %s' % (self.response.statusCode, self.response.statusMessage, errorMessage))
        self.response.connectionClose = True
        self.appendResponseMessageBody(errorMessage)

    # prepares an 500 Internal Server Error response, showing the provided errorMessage
    def setInternalServerError(self, errorMessage):
        self.response.cgiHeaders = {}
        self.response.statusCode = 500
        self.response.statusMessage = 'Internal Server Error'
        self.response.contentType = 'text/plain'
        self.response.contentLength = len(errorMessage)
        self.generateResponseHeaderMessage()
        self.logError('%i %s: %s' % (self.response.statusCode, self.response.statusMessage, errorMessage))
        self.response.connectionClose = True
        self.appendResponseMessageBody(errorMessage)

    # sends the response message to the client
    # returns true when the whole message was sent
    def flushResponseToClient(self):
        try:
            byteswritten = self.connection.send(self.response.message)
            self.response.message = self.response.message[byteswritten:]
            return len(self.response.message) == 0
        except:
            self.response.connectionClose = True
            return True


    # sends a pickled request/response wrapper object to the listener process
    # if closeConnection is set, that means that the connection will be closed after sending
    def flushResponseToListener(self, closeConnection=False):
        try:
            self.response.connectionClose = closeConnection
            # ofProcessor acts as a message queue if an output filter is specified
            # it accumulates the response body data, to be sent in one go to the filter
            if self.ofProcessor.execute():
                self.connection.send(self.pickle())
            self.response.flushed = True
            self.response.message = ''
            if closeConnection:
                self.connection.close()
                self.connectionClosed = True
        except:
            self.connection.close()
```

httprequest.py: HTTP/CGI parser and request handler

```
            self.connectionClosed = True


    # processes the request, i.e. determines whether a CGI script or a normal resource was accessed
    def process (self):
        # check if resource is inside the documentroot (jail)
        if self.isJailedInto(self.config.virtualHosts[self.request.virtualHost]['documentroot'], self.request.file
path):
            # determine chain of matching directories
            self.determineDirectoryChain()

            # check whether request is a CGI request, check documentroot and file existance
            typ = self.checkRequest()

            if typ == -1:
                self.processCGI()
            elif typ == -2:
                self.processDocument()
            else:
                self.sendError(typ)
        else:
            self.sendError(403,'Not allowed to access resource outside documentroot')


    # processes a normal resource request
    def processDocument(self):
        try:
            # privilege separation
            self.removePrivileges()
            self.response.contentType = self.determineContentType()
            self.response.contentLength = os.path.getsize(self.request.filepath)
            self.generateResponseHeaderMessage()
            # HEAD request must not have a response body, no need to access file
            if self.request.method != 'HEAD':
                self.accessFile(self.request.filepath)
            else:
                self.flushResponseToListener(True)
        except:
            self.sendError(500)


    # accesses a resource and sends the content back to the listener in chunks of data, i.e. not all at once
    # at the last "flush" the connection to the listener will be closed
    def accessFile(self, filename):
        f = file(filename,'r')

        data = f.read(self.config.configurations['socketbuffersize'])
        nextData = f.read(self.config.configurations['socketbuffersize'])
        while nextData and not self.connectionClosed:
            self.response.message = self.response.message + data
            # flush data part to listener and keep connection open
            self.flushResponseToListener()
            data = nextData
            nextData = f.read(self.config.configurations['socketbuffersize'])
        self.response.message = self.response.message + data
        # flush last data part to listener and close connection
        self.flushResponseToListener(True)
        f.close()


    def removePrivileges(self):
        st = os.stat(self.request.filepath)
        # don't remove privileges if process has already limited privileges
        if os.getuid() == 0:
            # if file is owned by root try to access is as default user
            if st.st_uid == 0:
                # default user
                os.setgid(self.config.configurations['group'])
                os.setuid(self.config.configurations['user'])
            else:
                # file owner user
                os.setgid(st.st_gid)
                os.setuid(st.st_uid)


    # processes a CGI script request
    def processCGI(self):
        try:
            self.removePrivileges()

            # check whether resource is an executable file (if no cgi executor set)
                if self.request.cgiExecutor == None and not os.access(self.request.filepath, os.X_OK):
                self.sendError(500,'CGI Script is not accessible/executable')
                return

            # generate environment variables for the CGI script
            self.generateCGIEnvironment()

            # execute cgi script - abort timeout of n seconds
            status = CGIExecutor(self).execute()

            # if execution was successful and no error was sent already
            if status == -1:
                self.sendError(500,'CGI Script aborted because of timeout')
```

```
        except:
            # Exception raised by the CGI executor
            self.sendError(500,'CGI script execution aborted')



    # checks whether the CGI response contained the Location header and it is a local redirect response
    # returns true if that is the case, otherwise false
    # additionally it monitors eventual endless loops that might occur if a cgiscript forwards to itself
    def checkReprocess(self):
        #Location flag set in CGI script
        if self.response.reprocess and self.response.getCGIHeader('Location') != None:
            self.requestNumber = self.requestNumber + 1
            # CGI local redirect response (RFC 6.2.2)
            self.parseURI(self.response.getCGIHeader('Location'))
            self.determineFilepath()
            # check for too many recursions
            if self.requestNumber > self.config.configurations['cgirecursionlimit']:
                self.setInternalServerError('Recursion in CGI script')
                return False
            return True
        else:
            return False


    # parses the headers of the CGI script
    # returns the pair (success,cgiBody)
    def parseCGIHeaders(self,document):
        document = document.lstrip()
        cgiBody = ''

        # determine end of line character (RFC says \n, but some implementations do \r\n)
        separator = '\n'
        pos = document.find('\n\n')
        posRN = document.find('\r\n\r\n')
        if pos == -1 or posRN != -1 and pos > posRN:
            pos = posRN
            separator = '\r\n'

        header = document[:pos]
        body = document[pos+len(separator)*2:]
        # parse header
        lines = header.split(separator)
        for line in lines:
            line = line.strip()
            line = re.sub('\s{2,}', ' ', line)
            pos = line.find(':')
            if pos <= 0 or pos >= len(line)-1:
                self.sendError(500,'Bad Header in CGI response')
                return (False,'')
            key = line[0:pos].strip()
            value = line[pos+1:len(line)].strip()
            self.response.setCGIHeader(key,value)

        if len(self.response.cgiHeaders) == 0:
            self.sendError(500,'CGI Script has no headers')
            return (False,'')

        location = self.response.getCGIHeader('Location')
        if location == None:
            # document response (RFC: 6.2.1)
            if body != None and body != '':
                if self.response.getCGIHeader('Content-Type') == None:
                    # content-type must be specified
                    self.sendError(500,'CGI Script must specify content type')
                    return (False,'')

                cgiBody = body

            # check for status header field
            if self.response.getCGIHeader('Status') != None:
                s = re.match(r'(\d+) (.*)',self.response.getCGIHeader('Status'),re.DOTALL)
                if s != None:
                    self.response.statusCode = int(s.group(1))
                    self.response.statusMessage = s.group(2)

        else:
            # redirect response
            if location.startswith('/'):
                # local redirect response (RFC: 6.2.2)
                self.response.reprocess = True
                newEnv = {}
                if self.request.cgiPathInfo != None:
                    newEnv['REDIRECT_URL'] = self.request.filepath[len(self.config.virtualHosts[self.request.vi
rtualHost]['documentroot']):] + self.request.cgiPathInfo
                else:
                    newEnv['REDIRECT_URL'] = self.request.filepath[len(self.config.virtualHosts[self.request.vi
rtualHost]['documentroot']):]
                newEnv['REDIRECT_STATUS'] = str(self.response.statusCode)
                # rename CGI environment variables
                for key in self.request.cgiEnv.keys():
                    newEnv['REDIRECT_'+key] = self.request.cgiEnv[key]
                self.request.cgiEnv = newEnv
            else:
```

```
                # client redirect response (RFC: 6.2.3, 6.2.4)
                self.response.statusCode = 302
                self.response.statusMessage = 'Found'
                self.response.setHeader('Location',location)

                if body != None and body != '':
                    if self.response.getCGIHeader('Content-Type') == None:
                        # content-type must be specified
                        self.sendError(500,'CGI Script must specify content type')
                        return (False,'')
                    # success
                    cgiBody = body

        return (True,cgiBody)


# This class provides an execution environment to the CGI script, which monitors the time it takes and aborts th
e script if it takes too long
class CGIExecutor():

    def __init__ (self, request):
        # Script process
        self.process = None
        # HttpRequest object
        self.request = request


    # executes the CGI script in a thread, which creates a new process that executes the requested scriptfile
    # the thread will cause the process to terminate after a timeout
    def execute (self):

        # executed in a separate thread
        def cgiThread():
            args = [self.request.request.filepath]
            if self.request.request.cgiExecutor != None:
                # use executor to run script
                args = [self.request.request.cgiExecutor,self.request.request.filepath]

            # creates a new process, running the script
            try:
                    self.process = subprocess.Popen(args,stdout=subprocess.PIPE,stdin=subprocess.PIPE,stderr=sub
process.PIPE,env=self.request.request.cgiEnv)

                # eventual POST data goes to stdin
                if self.request.request.body != '':
                    self.process.stdin.write(self.request.request.body)

                # fetch response blockwise and flush to listener
                out = self.process.stdout.read(self.request.config.configurations['socketbuffersize'])
                tmp = ''
                headerParsed = False
                success = True
                while out != '':
                    nextOut = self.process.stdout.read(self.request.config.configurations['socketbuffersize'])

                    if not headerParsed:
                        tmp = tmp + out
                        m = re.match(r'((.+)(\r\n\r\n|\n\n))(.*)',tmp,re.DOTALL)
                        if m != None:
                            headerParsed = True
                            success, cgiBody = self.request.parseCGIHeaders(tmp)
                            if success:
                                self.request.generateResponseHeaderMessage()
                                self.request.appendResponseMessageBody(cgiBody)
                                self.request.flushResponseToListener(nextOut == '')
                    else:
                        if success:
                            self.request.appendResponseMessageBody(out)
                            self.request.flushResponseToListener(nextOut == '')

                    out = nextOut

                if not self.request.response.flushed:
                    self.request.sendError(500,'Syntax Error in CGI Response')

                errorData = self.process.communicate()[1]

                # if some data is available on standarderror, log to errorlog
                if errorData.strip() != '':
                    self.request.logError(errorData)
            except:
                pass

        thread = threading.Thread(target=cgiThread)
        thread.start()
        thread.join(self.request.config.configurations['cgitimeout'])
        # if thread is still alive after timeout means that the script took to long
        if thread.is_alive():
            self.process.terminate()
            thread.join()
            return -1

        return 0
```

httprequest.py: HTTP/CGI parser and request handler

```
# executes one filter after the other
class OutputFilterProcessor:

    def __init__(self, request):
        self.request = request
        # response message
        self.message = ''
        # <directory> that matched the request
        self.outputFilterDirectory = None
        # current filter of the filterchain
        self.currentFilter = None
        # current filterprocess
        self.process = None

    # returns just the body of the response message
    def getBody(self):
        pos = self.message.find('\r\n\r\n')
        if pos == -1:
            return ''
        else:
            return self.message[pos+4:]

    # sets just the body of the response message
    def setBody(self, body):
        pos = self.message.find('\r\n\r\n')
        if pos == -1:
            return
        else:
            self.message = self.message[:pos+4] + body

    # starts the output filter processing, or just returns if there is no filter specified
    def execute(self):
        if self.request.response.statusCode >= 400 or self.outputFilterDirectory == None:
            return True

        # run filter in a separate thread, so it can be killed after a timeout (if it takes too long)
        def runFilter():
            try:
                script = self.request.config.virtualHosts[self.request.request.virtualHost]['extfilterdefine'][self.c
urrentFilter]
                self.process = subprocess.Popen(script,stdout=subprocess.PIPE,stdin=subprocess.PIPE,stderr=subpr
ocess.PIPE)
                # response body goes to stdin
                self.process.stdin.write(self.getBody())
                # the response is on standardoutput
                body, errorData = self.process.communicate()
                self.setBody(body)
                # if some data is available on standarderror, log to errorlog
                if errorData != None and errorData != '':
                    self.request.logError(errorData.replace('\n',''))
            except:
                self.request.sendError(500,'Error executing filter '+self.currentFilter)

        if self.request.response.connectionClose:
            self.message = self.message + self.request.response.message
            # all data received
            # run one filter after the other
            for f in self.request.config.virtualHosts[self.request.request.virtualHost]['directory'][self.outputFi
lterDirectory]['setoutputfilter']:
                self.currentFilter = f
                thread = threading.Thread(target=runFilter)
                thread.start()
                thread.join(self.request.config.configurations['cgitimeout'])
                # if thread is still alive after timeout means that the script took to long
                if thread.is_alive():
                    self.process.terminate()
                    thread.join()
                    self.request.sendError(500,'Filter aborted because of timeout '+self.currentFilter)

                if self.request.response.statusCode >= 400:
                    # break if an error occurred
                    return False

            self.request.response.message = self.message
            return True
        else:
            # more data to receive
            self.message = self.message + self.request.response.message
            return False
```

config.py: configuration file parser

```
import os, pwd, grp
from os import path, sep
import logging
import re

# filter classes, that help logging error and access logs into different files
class ErrorFilter(logging.Filter):
    def filter(self, record):
        return record.levelno == logging.ERROR or record.levelno == logging.WARN

class InfoFilter(logging.Filter):
    def filter(self, record):
        return record.levelno == logging.INFO

# this class retrieves the servers main log file, parses it and then retrieves all virtualhost configuration fil
es and also parses them.
# additionally it initializes the logger
class SwsConfiguration:

    # name of the main log file
    MAIN_CONFIG_FILE = 'sws.conf'
    # name of the folder, containing a .conf file for each virtualhost
    SITES_ENABLED_FOLDER = 'sites-enabled'

    def __init__(self, configFolder):
        # folder containing the main configuration file and the sites-enabled folder
        self.configFolder = path.abspath(configFolder)
        # path to the configuration file
        self.configFile = path.abspath(self.configFolder + sep + SwsConfiguration.MAIN_CONFIG_FILE)
        # object containing the server configuration
        self.configurations = {
            # port on which the server listens
            'listen':80,
            # address on which the server listens
            'host':'',
            # UNIX socket file for communication between processes
            'communicationsocketfile':None,
            # listener user
            'user':None,
            # listener group
            'group':None,
            # whether the server performs hostname lookups (turn off for performance reasons)
            'hostnamelookups':False,
            # default content type, the server delivers if the content type cannot be determined
            'defaulttype':None,
            # timeout after which cgi scripts get aborted
            'cgitimeout':30,
            # number of consecutive local redirects
            'cgirecursionlimit':10,
            # size of the listen queue for incoming connections
            'listenqueuesize':10,
            # size of the communication buffer
            'socketbuffersize':8192,
            # root for errordocuments
            'errordocumentroot':None,
            # error documents
            'errordocument':{
                403:{'msg':'Forbidden','defaulttxt':'Status 403 - Forbidden. You are not allowed to access this resource.','file':None},
                404:{'msg':'Not Found','defaulttxt':'Status 404 - File Not Found','file':None},
                500:{'msg':'Internal Server Error','defaulttxt':'Status 500 - Internal Server Error','file':None}
            },
            # logfile for error messages
            'errorlogfile':None,
            # logfile for access messages
            'accesslogfile':None,
            # serverside file-extension / content-type associations
            'addtype':{}
        }
        # obejct containing the configurations for every virtualhost
        self.virtualHosts = {}
        # name of the default virtualhost, that is applied if no virtualhost matches a given hostname
        self.defaultVirtualHost = None
        # initialize logger to critical, so that no logging takes place if the log file can not be determined
        logging.getLogger('sws').setLevel(logging.CRITICAL)


    # create a new virtualhost object
    def initVHost (self, vHost):
        # maintain standard errordocuments
        errorDocs = {}
        for err in self.configurations['errordocument'].keys():
            errorDocs[err] = self.configurations['errordocument'][err]['file']

        self.virtualHosts[vHost] = {
            'serveradmin':'',
            'servername':None,
            'serveralias':[],
            'documentroot':None,
            'errorlogfile':self.configurations['errorlogfile'],
            'accesslogfile':self.configurations['accesslogfile'],
            'errordocumentroot':self.configurations['errordocumentroot'],
            'errordocument':errorDocs,
            'directory':{'/':self.initDirectory()},
            'extfilterdefine':{}
```

```
        }

    # create a new directory object
    def initDirectory(self):
        return  {
            'directoryindex':[],
            'cgihandler':[],
            'setoutputfilter':[],
            'addheader':{},
            'addtype':{},
            # subfolders should(n't) inherit directive from parent folder
            'stopinheritance':{
                'directoryindex':False,
                'cgihandler':False,
                'setoutputfilter':False,
                'addheader':False,
                'addtype':False
            }
        }


    # get data out of the configuration file
    def readConfigFile(self, configFile):
        configLines = ''
        try:
            f = open (configFile,'r')
            line = 'init'
            while line != '':
                line = f.readline()
                if line.strip().startswith('#') or line.strip() == '':
                    continue
                configLines = configLines + line
            return (True, configLines)
        except:
            return (False, 'Cannot read configuration file '+configFile)


    # initialize a logger (either global or optional loggers for every virtualhost)
    def initLogger(self, name, errorFile, accessFile):
        logger = logging.getLogger(name)
        logger.setLevel(logging.INFO)

        errorHandler = logging.FileHandler(errorFile)
        errorHandler.addFilter(ErrorFilter())
        errorFormatter = logging.Formatter('%(message)s')
        errorHandler.setFormatter(errorFormatter)

        accessHandler = logging.FileHandler(accessFile)
        accessHandler.addFilter(InfoFilter())
        accessFormatter = logging.Formatter('%(message)s')
        accessHandler.setFormatter(accessFormatter)

        logger.addHandler(errorHandler)
        logger.addHandler(accessHandler)


    # start the configuration files parsing process
    def parseFile(self):
        # parse main config file

        if not os.path.isdir(self.configFolder):
            return (False, 'Configuration folder '+self.configFolder+' not found',10)
        if not os.path.isfile(self.configFile):
            return (False, 'Configuration file '+self.configFile+' not found',11)

        success, configLines = self.readConfigFile(self.configFile)
        if not success:
            return (False,configLines,12)

        lines = configLines.splitlines()
        for line in lines:

            # check line by line
            line = line.strip()
            fields = line.split()

            if len(fields) < 2:
                return (False, 'Syntax error in configuration directive: '+line,13)

            directive = fields[0].lower()

            if directive not in self.configurations.keys():
                return (False, 'Unknown configuration directive: '+line,14)

            if directive in ['errordocument','addtype'] and len(fields) != 3:
                return (False, 'Syntax error in configuration directive: '+line,15)

            if directive not in ['errordocument','addtype'] and len(fields) != 2:
                return (False, 'Syntax error in configuration directive: '+line,16)

            # integer directives
            if directive in ['listen','cgitimeout','listenqueuesize','socketbuffersize','cgirecursionlimit']:
                try:
                    value = int(fields[1])
```

config.py: configuration file parser

```
                if value <= 0:
                    return (False, 'Value less or equal to 0 not supported in configuration directive: '+line,43)
                self.configurations[directive] = value
                continue
            except:
                return (False, 'Type error in configuration directive: '+line,17)

        # boolean directives
        if directive in ['hostnamelookups']:
            if fields[1].lower() == 'on':
                self.configurations[directive] = True
            elif fields[1].lower() == 'off':
                self.configurations[directive] = False
            else:
                return (False, 'Type error in configuration directive: '+line,18)
            continue

        # errordocument
        if directive in ['errordocument']:
            code = -1
            try:
                code = int(fields[1])
            except:
                return (False, 'Type error in code of errordocument directive: '+line,19)
            if not code in self.configurations[directive].keys():
                return (False, 'Error code not supported by server: '+line,20)
            self.configurations['errordocument'][code]['file'] = fields[2]
            continue

        # file
        if directive in ['communicationsocketfile','errorlogfile','accesslogfile']:
            filepath = os.path.abspath(fields[1])
            pos = filepath.rfind(sep)
            if not os.path.isdir(filepath[:pos]):
                return (False, 'Folder does not exist: '+filepath[:pos],21)
            self.configurations[directive] = filepath
            continue

        # addtype
        if directive in ['addtype']:
            extension = fields[1]
            if not extension.startswith('.'):
                extension = '.' + extension
            if '/' in extension:
                return (False,'Syntax error in configuration directive: '+line,49)
            self.configurations[directive][extension] = fields[2]
            continue

        # directory
        if directive in ['errordocumentroot']:
            if not os.path.isdir(os.path.abspath(fields[1])):
                return (False, 'Folder does not exist: '+line,22)
            self.configurations[directive] = os.path.abspath(fields[1])
            continue

        # user
        if directive in ['user']:
            try:
                pw = pwd.getpwnam(fields[1])
                self.configurations[directive] = int(pw.pw_uid)
            except:
                try:
                    pw = pwd.getpwuid(int(fields[1]))
                    self.configurations[directive] = int(pw.pw_uid)
                except:
                    return (False, 'User does not exist: '+line,23)
            continue

        # group
        if directive in ['group']:
            try:
                gr = grp.getgrnam(fields[1])
                self.configurations[directive] = int(gr.gr_gid)
            except:
                try:
                    gr = grp.getgrgid(int(fields[1]))
                    self.configurations[directive] = int(gr.gr_gid)
                except:
                    return (False, 'Group does not exist: '+line,24)
            continue

        # string directive
        self.configurations[directive] = fields[1]

    for directive in self.configurations.keys():
        if self.configurations[directive] == None:
            return (False, 'Mandatory directive not specified: '+directive,25)


    # init main logger
    logging.getLogger('sws').setLevel(logging.INFO)
    self.initLogger('sws',self.configurations['errorlogfile'],self.configurations['accesslogfile'])
```

config.py: configuration file parser

```
        # parse virtualhosts

        # check if sites enabled folder exists
        sitesEnabled = os.path.abspath(self.configFolder + sep + SwsConfiguration.SITES_ENABLED_FOLDER)
        if not os.path.isdir(sitesEnabled):
            return (False, 'Folder does not exists: '+sitesEnabled,26)

        # process every .conf file in the sites-enabled folder
        for filename in os.listdir(sitesEnabled):
            vHost = os.path.abspath(sitesEnabled + sep + filename)
            # skip directories
            if not os.path.isfile(vHost) or not vHost.endswith('.conf'):
                continue

            success, configLines = self.readConfigFile(vHost)
            if not success:
                return (False,configLines)

            self.initVHost(vHost)
            lines = configLines.splitlines()

            curDirectory = '/'
            directoryOpen = False

            for line in lines:

                # process every line of a configuration file
                line = line.strip()

                # check for <directory> directive
                m = re.match(r'<[Dd][Ii][Rr][Ee][Cc][Tt][Oo][Rr][Yy]\s+"(.+)"\s*>',line,re.DOTALL)
                if m != None and self.virtualHosts[vHost]['documentroot'] == None:
                    return (False,'Please specifiy Documentroot before: '+line,27)

                if m != None and directoryOpen:
                    return (False,'Nesting of <Directory> directives not allowed: '+line,28)

                if m != None:
                    directoryOpen = True
                    curDirectory = path.abspath(self.virtualHosts[vHost]['documentroot'] + sep + m.group(1))[len(s
elf.virtualHosts[vHost]['documentroot']):]
                    if curDirectory == '':
                        curDirectory = '/'
                    if curDirectory not in self.virtualHosts[vHost]['directory'].keys():
                        self.virtualHosts[vHost]['directory'][curDirectory] = self.initDirectory()
                    continue


                m2 = re.match(r'</[Dd][Ii][Rr][Ee][Cc][Tt][Oo][Rr][Yy]>',line,re.DOTALL)
                if m2 != None and directoryOpen:
                    directoryOpen = False
                    curDirectory = '/'
                    continue

                fields = line.split()

                if len(fields) < 1:
                    return (False, 'Syntax error in configuration directive: '+line,29)

                directive = fields[0].lower()

                if directoryOpen and directive not in ['directoryindex','cgihandler','setoutputfilter','addheader','stopinheritance
','addtype']:
                    return (False,'Directive not allowed in <Directory>: '+directive,30)

                # defaultvirtualhost
                if directive in ['defaultvirtualhost']:
                    if self.defaultVirtualHost == None:
                        self.defaultVirtualHost = vHost
                        continue
                    else:
                        return (False,'Multiple Default VirtualHosts',31)

                if directive not in self.virtualHosts[vHost].keys() and directive not in self.virtualHosts[vHost
]['directory']['/'].keys():
                    return (False, 'Unknown configuration directive: '+line,32)

                if len(fields) < 2:
                    return (False, 'Syntax error in configuration directive: '+line,33)

                if directive in ['errordocument','addtype'] and len(fields) != 3:
                    return (False, 'Syntax error in configuration directive: '+line,34)

                if directive not in ['errordocument','directoryindex','serveralias','cgihandler','addtype','extfilterdefine','addheader
','stopinheritance'] and len(fields) != 2:
                    return (False, 'Syntax error in configuration directive: '+line,35)

                if directive in ['cgihandler'] and len(fields) > 3:
                    return (False, 'Syntax error in configuration directive: '+line,47)


                if directive in ['addheader']:
                    header = fields[1].replace('"','')
                    args = ''
```

config.py: configuration file parser

```
                    i = 2
                    while i < len(fields):
                        if i != 2:
                            args = args + ' ' + fields[i]
                        else:
                            args = args + fields[i]
                        i = i + 1
                    m = re.match(r'"(.+)"',args,re.DOTALL)
                    if m == None or header == '':
                        return (False, 'Syntax error in configuration directive: '+line,54)
                    self.virtualHosts[vHost]['directory'][curDirectory][directive][header] = m.group(1)
                    continue

                if directive in ['extfilterdefine']:
                    args = ''
                    i = 2
                    while i < len(fields):
                        if i != 2:
                            args = args + ' ' + fields[i]
                        else:
                            args = args + fields[i]
                        i = i + 1
                    m = re.match(r'cmd\s*=\s*"(.+)"',args,re.DOTALL)
                    if m == None:
                        return (False, 'Syntax error in configuration directive: '+line,50)
                    if fields[1] in self.virtualHosts[vHost][directive].keys():
                        return (False, 'Duplicate filter definition: '+line,51)
                    self.virtualHosts[vHost][directive][fields[1]] = m.group(1).split()
                    continue

                if directive in ['setoutputfilter']:
                    if len(self.virtualHosts[vHost]['directory'][curDirectory][directive]) != 0:
                        return (False, 'Not allowed to define multiple filter chains in one directory: '+line,52)
                    filters = fields[1].split(';')
                    for f in filters:
                        if f not in self.virtualHosts[vHost]['extfilterdefine'].keys():
                            return (False, 'Unknown filter: '+f,53)
                        self.virtualHosts[vHost]['directory'][curDirectory][directive].append(f)
                    continue

                # multiple values
                if directive in ['serveralias']:
                    first = False
                    for field in fields:
                        if not first:
                            first = True
                            continue
                        self.virtualHosts[vHost][directive].append(field)
                    continue

                # possible multiple values
                if directive in ['stopinheritance']:
                    first = False
                    for field in fields:
                        if not first:
                            first = True
                            continue
                        field = field.lower()
                        if field not in self.virtualHosts[vHost]['directory'][curDirectory][directive].keys() and
field not in ['all']:
                            return (False, 'Syntax error in configuration directive: '+line,55)
                        if field == 'all':
                            for k in self.virtualHosts[vHost]['directory'][curDirectory][directive].keys():
                                self.virtualHosts[vHost]['directory'][curDirectory][directive][k] = True
                        else:
                            self.virtualHosts[vHost]['directory'][curDirectory][directive][field] = True
                    continue


                # multiple values in <directory>
                if directive in ['directoryindex']:
                    first = False
                    for field in fields:
                        if not first:
                            first = True
                            continue
                        self.virtualHosts[vHost]['directory'][curDirectory][directive].append(field)
                    continue

                # cgihandler
                if directive in ['cgihandler']:
                    extension = fields[1]
                    if not extension.startswith('.'):
                        extension = '.' + extension
                    if '/' in extension:
                        return (False,'Syntax error in configuration directive: '+line,44)
                    executor = None
                    if len(fields) == 3:
                        executor = os.path.abspath(fields[2])
                        if not os.path.isfile(executor):
                            return (False,'CGI Executor could not be found: '+executor,45)
                        if not os.access(executor,os.X_OK):
                            return (False,'CGI Executor is not an executable file: '+executor,46)
```

config.py: configuration file parser

```
                    self.virtualHosts[vHost]['directory'][curDirectory][directive].append({'extension':extension,'exec
utor':executor})
                    continue

                # directory
                if directive in ['documentroot','errordocumentroot']:
                    if not os.path.isdir(os.path.abspath(fields[1])):
                        return (False, 'Folder does not exist: '+line,36)
                    self.virtualHosts[vHost][directive] = os.path.abspath(fields[1])
                    continue

                # errordocument
                if directive in ['errordocument']:
                    code = -1
                    try:
                        code = int(fields[1])
                    except:
                        return (False, 'Type error in code of errordocument directive: '+line,37)
                    if not code in self.configurations[directive].keys():
                        return (False, 'Error code not supported by server: '+line,38)
                    self.virtualHosts[vHost]['errordocument'][code] = fields[2]
                    continue

                # addtype
                if directive in ['addtype']:
                    extension = fields[1]
                    if not extension.startswith('.'):
                        extension = '.' + extension
                    if '/' in extension:
                        return (False,'Syntax error in configuration directive: '+line,48)
                    self.virtualHosts[vHost]['directory'][curDirectory][directive][extension] = fields[2]
                    continue

                # file
                if directive in ['errorlogfile','accesslogfile']:
                    filepath = os.path.abspath(fields[1])
                    pos = filepath.rfind(sep)
                    if not os.path.isdir(filepath[:pos]):
                        return (False, 'Folder does not exist: '+filepath[:pos],39)
                    self.virtualHosts[vHost][directive] = filepath
                    continue

                # string directive
                self.virtualHosts[vHost][directive] = fields[1]


            if directoryOpen:
                return (False, 'Missing </Directory> directive',40)

        if len(self.virtualHosts) == 0:
            return (False, 'No VirtualHost specified',41)

        if self.defaultVirtualHost == None:
            return (False, 'No DefaultVirtualHost specified',42)


        # check for mandatory directives
        for vHost in self.virtualHosts.keys():
            # set logger
            self.initLogger(vHost,self.virtualHosts[vHost]['errorlogfile'],self.virtualHosts[vHost]['accesslogfile'])

        # check for mandatory directives
        for vHost in self.virtualHosts.keys():
            for directive in self.virtualHosts[vHost].keys():
                if self.virtualHosts[vHost][directive] == None:
                    return (False, 'Mandatory directive ('+directive+') not specified in VirtualHost: '+vHost,43)

        return (True,'Parsing OK',0)
```

daemon.py: realises the daemon functionality (open source, public domain)

```
import sys, os, time, atexit, stat
from signal import SIGTERM

# This class implements a daemon. It is open source and public domain.
# And has been modified for our purposes.
class Daemon:
    """
    A generic daemon class.
    Usage: subclass the Daemon class and override the run() method
    """
    def __init__(self, pidfile, configfile, stdin='/dev/null', stdout='/dev/null', stderr='/dev/null'):
        self.stdin = stdin
        self.stdout = stdout
        self.stderr = stderr
        self.pidfile = pidfile
        self.configfile = configfile

    def daemonize(self):
        """
        do the UNIX double-fork magic, see Stevens' "Advanced
        Programming in the UNIX Environment" for details (ISBN 0201563177)
        http://www.erlenstar.demon.co.uk/unix/faq_2.html#SEC16
        """
        try:
            pid = os.fork()
            if pid > 0:
                # exit first parent
                sys.exit(0)
        except OSError, e:
            sys.stderr.write("fork #1 failed: %d (%s)\n" % (e.errno, e.strerror))
            sys.exit(1)

        # decouple from parent environment
        os.chdir("/")
        os.setsid()
        os.umask(0)

        # do second fork
        try:
            pid = os.fork()
            if pid > 0:
                # exit from second parent
                sys.exit(0)
        except OSError, e:
            sys.stderr.write("fork #2 failed: %d (%s)\n" % (e.errno, e.strerror))
            sys.exit(1)


        msg = self.initialize(self.configfile)
        if msg != None:
            sys.stderr.write("Error: %s\n" % msg)
            sys.exit(1)

        # redirect standard file descriptors
        sys.stdout.flush()
        sys.stderr.flush()
        si = file(self.stdin, 'r')
        so = file(self.stdout, 'a+')
        se = file(self.stderr, 'a+', 0)
        os.dup2(si.fileno(), sys.stdin.fileno())
        os.dup2(so.fileno(), sys.stdout.fileno())
        os.dup2(se.fileno(), sys.stderr.fileno())

        # write pidfile
        atexit.register(self.delpid)
        pid = str(os.getpid())
        file(self.pidfile,'w+').write("%s\n" % pid)

    def delpid(self):
        try:
            os.remove(self.pidfile)
        except:
            pass

    def start(self):
        """
        Start the daemon
        """
        # Check for a pidfile to see if the daemon already runs
        try:
            pf = file(self.pidfile,'r')
            pid = int(pf.read().strip())
            pf.close()
        except IOError:
            pid = None

        if pid:
            message = "pidfile %s already exist. Daemon already running?\n"
            sys.stderr.write(message % self.pidfile)
            sys.exit(1)

        # Start the daemon
        self.daemonize()
        self.run()
```

daemon.py: realises the daemon functionality (open source, public domain)

```
    def stop(self):
        """
        Stop the daemon
        """
        # Get the pid from the pidfile
        try:
            pf = file(self.pidfile,'r')
            pid = int(pf.read().strip())
            pf.close()
        except IOError:
            pid = None

        if not pid:
            message = "pidfile %s does not exist. Daemon not running?\n"
            sys.stderr.write(message % self.pidfile)
            return # not an error in a restart

        # Try killing the daemon process
        try:
            while 1:
                os.kill(pid, SIGTERM)
                time.sleep(0.1)
        except OSError, err:
            err = str(err)
            if err.find("No such process") > 0:
                if os.path.exists(self.pidfile):
                    os.remove(self.pidfile)
            else:
                print str(err)
                sys.exit(1)

    def restart(self):
        """
        Restart the daemon
        """
        self.stop()
        self.start()

    def run(self):
        """
        You should override this method when you subclass Daemon. It will be called after the process has been
        daemonized by start() or restart().
        """

    def initialize(self):
        """
        You should override this method when you subclass Daemon. It will be called before the process will be
        daemonized by start() or restart().
        """
```

main.py: PyUnit main class for running all unit tests

```
#!/usr/bin/python -B

import unittest

from configtestcase import ConfigTestCase
from servertestcase import ServerTestCase

# Run's both unit test suites, i.e., tests the configuration file parser and the web-server functionalities
if __name__ == "__main__":
    suite1 = unittest.TestLoader().loadTestsFromTestCase(ConfigTestCase)
    suite2 = unittest.TestLoader().loadTestsFromTestCase(ServerTestCase)
    unittest.main()
```

servertestcase.py: PyUnit test cases for testing web–server functionalities

```
#!/usr/bin/python -B

import unittest
import sys
sys.path.append('../code')
import config
import httplib

# This class provides unit test cases for testing the webserver functionalities.
# Note, that on different systems some tests might fail,
# because they are configured for my execution environment (users, privileges, etc.).
class ServerTestCase (unittest.TestCase):

    PORT = 81
    METHODS = ['GET','POST','HEAD']

    def testLocalhost(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/')
            response = connection.getresponse()
            assert response.status == 200
            data = response.read()
            data = data.strip()
            if method == 'HEAD':
                assert data == ''
            else:
                assert data == 'success'
            connection.close()

    def testLocalhostForbidden(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # break out of documentroot
            connection.request(method,'../127.0.0.1/index.html')
            response = connection.getresponse()
            assert response.status == 403
            connection.close()


    def testLocalhostNotFound(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # try to access not existing file
            connection.request(method,'index_notfound.html')
            response = connection.getresponse()
            assert response.status == 404
            connection.close()


    def testLocalhostMethodNotSupported(self):
        connection = httplib.HTTPConnection('localhost', self.PORT)
        connection.connect()
        # try invalid request method
        connection.request('NOTSUPPORTED','/')
        response = connection.getresponse()
        assert response.status == 400
        connection.close()


    def testLocalhostRequestBody(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/','variable=100',{'myheader':123})
            response = connection.getresponse()
            assert response.status == 200
            connection.close()


    def testLocalhostCGIExecutor(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # owner of the file must be set to stefan
            connection.request(method,'/cgi-bin/executor.sh')
            response = connection.getresponse()
            assert response.status == 200
            data = response.read()
            data = data.strip()
            if method == 'HEAD':
                assert data == ''
            else:
                assert data == 'stefan'
            connection.close()


    def test127001CGIExecutor(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('127.0.0.1', self.PORT)
            connection.connect()
```

```
            # owner of the file must be set to www-data
            connection.request(method,'/cgi-bin/executor.sh')
            response = connection.getresponse()
            assert response.status == 200
            data = response.read()
            data = data.strip()
            if method == 'HEAD':
                assert data == ''
            else:
                assert data == 'www-data'
            connection.close()


    def testLocalhostRequestBodyCGI(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/cgi-bin/post.pl?v2=200','v1=100')
            response = connection.getresponse()
            data = response.read()
            data = data.strip()
            assert response.status == 200

            if method == 'HEAD':
                assert data == ''
            elif method == 'POST':
                assert data == 'v1 => 100'
            else:
                assert data == 'v2 => 200'
            connection.close()


    def testLocalhostRequestBodyCGIPHP(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/cgi-bin/php?v1=4','v2=5',{'content-type':'application/x-www-form-urlencoded'})
            response = connection.getresponse()
            data = response.read()
            data = data.strip()
            assert response.status == 200
            if method == 'HEAD':
                assert data == ''
            elif method == 'POST':
                assert data == '9'
            else:
                assert data == '4'
            connection.close()

    def testLocalhostCGIRedirect1(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # redirect to 127.0.0.1
            connection.request(method,'/cgi-bin/loc.pl')
            response = connection.getresponse()
            assert response.status == 302
            assert response.getheader('Location') == 'http://127.0.0.1:81/'
            connection.close()

    def testLocalhostCGIRedirect2(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # redirect to /
            connection.request(method,'/cgi-bin/loc2.pl')
            response = connection.getresponse()
            assert response.status == 200
            data = response.read()
            data = data.strip()
            if method == 'HEAD':
                assert data == ''
            else:
                assert data == 'success'
            connection.close()

    def testLocalhostCGIRedirect3(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # redirect to itself -> recursion
            connection.request(method,'/cgi-bin/loc3.pl')
            response = connection.getresponse()
            assert response.status == 500
            connection.close()


    def testLocalhostCGIForever(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # endless script -> abort (CGITimeout in configuration set to 1 sek)
            connection.request(method,'/cgi-bin/forever.sh')
            response = connection.getresponse()
```

```
            assert response.status == 500
            connection.close()


    def testLocalhostCGIByRootNoAccess(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # script owned by root and no access privileges for default user
            connection.request(method,'/cgi-bin/env.pl')
            response = connection.getresponse()
            assert response.status == 500
            connection.close()

    def testLocalhostCGIByRootAccess(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # script owned by root and access privileges for default user (stefan)
            connection.request(method,'/cgi-bin/executor2.sh')
            response = connection.getresponse()
            assert response.status == 200
            data = response.read()
            data = data.strip()
            if method == 'HEAD':
                assert data == ''
            else:
                assert data == 'stefan'
            connection.close()


    def testLocalhostCGINotExecutable(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            # script is not executable
            connection.request(method,'/cgi-bin/notexecutable.pl')
            response = connection.getresponse()
            assert response.status == 500
            connection.close()


    def testLocalhostCGIBadHeader(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/cgi-bin/badheader.pl')
            response = connection.getresponse()
            assert response.status == 500
            connection.close()


    def testLocalhostCGINoContentType(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/cgi-bin/nocontenttype.pl')
            response = connection.getresponse()
            assert response.status == 500
            connection.close()


    def testLocalhostCGINoCGI(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/cgi-bin/notcgi.html')
            response = connection.getresponse()
            assert response.status == 200
            data = response.read()
            data = data.strip()
            if method == 'HEAD':
                assert data == ''
            else:
                assert data == 'successnotcgi'
            connection.close()

    def testLocalhostFilter(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/filter/index.html')
            response = connection.getresponse()
            assert response.status == 200
            assert response.getheader('content-encoding') == 'gzip'
            connection.close()

    def testLocalhostEndlessFilter(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/filter2/index.html')
            response = connection.getresponse()
            assert response.status == 500
```

```
            connection.close()

    def testLocalhostFilterFileNotFound(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/filter2/index2.html')
            response = connection.getresponse()
            assert response.status == 404
            connection.close()

    def testLocalhostFilterScriptNotFound(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/filter3/index.html')
            response = connection.getresponse()
            assert response.status == 500
            connection.close()

    def testLocalhostFiltered(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/filter4/index.html')
            response = connection.getresponse()
            assert response.status == 200
            data = response.read()
            data = data.strip()
            if method == 'HEAD':
                assert data == ''
            else:
                assert data == 'filtered'
            connection.close()

    def testLocalhostFilterNotExecutable(self):
        for method in self.METHODS:
            connection = httplib.HTTPConnection('localhost', self.PORT)
            connection.connect()
            connection.request(method,'/filter5/index.html')
            response = connection.getresponse()
            assert response.status == 500
            connection.close()

if __name__ == "__main__":
    unittest.main()
```

configtestcase.py: PyUnit test cases for testing configuration file parser

```python
#!/usr/bin/python -B

import unittest
import sys
sys.path.append('../code')
import config
import httplib

# This class provides unit test cases for testing the configuration file parser.
# Note, that on different systems some tests might fail,
# because they are configured for my execution environment (users, privileges, etc.).
class ConfigTestCase (unittest.TestCase):

    CONFIG_FOLDER = '/home/stefan/sws/test/config'

    def testConfigFolderNotFound(self):
        testfolder = self.CONFIG_FOLDER + '/t0'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 10

    def testConfigMainFileNotFound(self):
        testfolder = self.CONFIG_FOLDER + '/t1'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 11

    def testConfigMainSyntaxError1(self):
        testfolder = self.CONFIG_FOLDER + '/t2'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 13

    def testConfigMainSyntaxError2(self):
        testfolder = self.CONFIG_FOLDER + '/t3'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 14

    def testConfigMainSyntaxError3(self):
        testfolder = self.CONFIG_FOLDER + '/t4'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 15

    def testConfigMainSyntaxError4(self):
        testfolder = self.CONFIG_FOLDER + '/t5'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 16

    def testConfigMainSyntaxError5(self):
        testfolder = self.CONFIG_FOLDER + '/t6'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 17

    def testConfigMainSyntaxError6(self):
        testfolder = self.CONFIG_FOLDER + '/t7'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 18

    def testConfigMainSyntaxError7(self):
        testfolder = self.CONFIG_FOLDER + '/t8'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 19

    def testConfigMainInvalidErrorCode(self):
        testfolder = self.CONFIG_FOLDER + '/t9'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 20

    def testConfigMainFolderNotFound1(self):
        testfolder = self.CONFIG_FOLDER + '/t10'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 21

    def testConfigMainFolderNotFound2(self):
        testfolder = self.CONFIG_FOLDER + '/t11'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 22

    def testConfigMainUserGroup1(self):
        testfolder = self.CONFIG_FOLDER + '/t12'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 23

    def testConfigMainUserGroup2(self):
        testfolder = self.CONFIG_FOLDER + '/t13'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 23

    def testConfigMainUserGroup3(self):
        testfolder = self.CONFIG_FOLDER + '/t14'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 24

    def testConfigMainUserGroup4(self):
        testfolder = self.CONFIG_FOLDER + '/t15'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 24
```

```
    def testConfigMainMandatoryMissing(self):
        testfolder = self.CONFIG_FOLDER + '/t16'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 25

    def testConfigSitesEnabledNotFound(self):
        testfolder = self.CONFIG_FOLDER + '/t17'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 26

    def testConfigNoVirtualHostsFound(self):
        testfolder = self.CONFIG_FOLDER + '/t18'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 41

    def testConfigNoDefaultVirtualHostFound(self):
        testfolder = self.CONFIG_FOLDER + '/t19'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 42

    def testConfigVHMandatoryDirectiveNotFound(self):
        testfolder = self.CONFIG_FOLDER + '/t20'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 43

    def testConfigVHDirectoryBeforeDocumentroot(self):
        testfolder = self.CONFIG_FOLDER + '/t21'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 27

    def testConfigVHDirectoryNesting(self):
        testfolder = self.CONFIG_FOLDER + '/t22'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 28

    def testConfigVHDirectiveNotAllowed(self):
        testfolder = self.CONFIG_FOLDER + '/t23'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 30

    def testConfigVHMultipleDefaultVirtualHosts(self):
        testfolder = self.CONFIG_FOLDER + '/t24'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 31

    def testConfigVHUnknownDirective(self):
        testfolder = self.CONFIG_FOLDER + '/t25'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 32

    def testConfigVHSyntaxError1(self):
        testfolder = self.CONFIG_FOLDER + '/t26'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 33

    def testConfigVHSyntaxError2(self):
        testfolder = self.CONFIG_FOLDER + '/t27'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 34

    def testConfigVHSyntaxError3(self):
        testfolder = self.CONFIG_FOLDER + '/t28'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 35

    def testConfigVHDirectoryNotFound(self):
        testfolder = self.CONFIG_FOLDER + '/t29'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 36

    def testConfigVHSyntaxError4(self):
        testfolder = self.CONFIG_FOLDER + '/t30'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 37

    def testConfigVHSyntaxError5(self):
        testfolder = self.CONFIG_FOLDER + '/t31'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 38

    def testConfigVHSyntaxError6(self):
        testfolder = self.CONFIG_FOLDER + '/t32'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 39

    def testConfigVHSyntaxError7(self):
        testfolder = self.CONFIG_FOLDER + '/t33'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 40

    def testConfigMainNegativeValueError7(self):
        testfolder = self.CONFIG_FOLDER + '/t34'
        c = config.SwsConfiguration(testfolder)
```

```
        assert c.parseFile()[2] == 43

    def testConfigVHWrongHandler(self):
        testfolder = self.CONFIG_FOLDER + '/t35'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 44

    def testConfigVHWrongExecutor(self):
        testfolder = self.CONFIG_FOLDER + '/t36'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 45

    def testConfigVHWrongExecutor2(self):
        testfolder = self.CONFIG_FOLDER + '/t37'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 46

    def testConfigVHWrongExecutor3(self):
        testfolder = self.CONFIG_FOLDER + '/t38'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 47

    def testConfigVHWrongAddType(self):
        testfolder = self.CONFIG_FOLDER + '/t39'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 49

    def testConfigMainWrongAddType(self):
        testfolder = self.CONFIG_FOLDER + '/t40'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 48

    def testConfigVHWrongFilter1(self):
        testfolder = self.CONFIG_FOLDER + '/t41'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 50

    def testConfigVHWrongFilter2(self):
        testfolder = self.CONFIG_FOLDER + '/t42'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 51

    def testConfigVHWrongFilter3(self):
        testfolder = self.CONFIG_FOLDER + '/t43'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 52

    def testConfigVHWrongFilter4(self):
        testfolder = self.CONFIG_FOLDER + '/t44'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 53

    def testConfigVHWrongHeader(self):
        testfolder = self.CONFIG_FOLDER + '/t45'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 54

    def testConfigVHWrongStopInheritation(self):
        testfolder = self.CONFIG_FOLDER + '/t46'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 55

    def testConfigFileOK(self):
        testfolder = self.CONFIG_FOLDER + '/t50'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[2] == 0

    def testConfigMainFile(self):
        testfolder = self.CONFIG_FOLDER + '/t51'
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[0] == True
        assert c.configurations['listen'] == 8080
        assert c.configurations['host'] == '127.0.0.1'
        assert c.configurations['user'] == 1000
        assert c.configurations['group'] == 33
        assert c.configurations['hostnamelookups'] == True
        assert c.configurations['defaulttype'] == 'text/plain'
        assert c.configurations['cgitimeout'] == 23
        assert c.configurations['cgirecursionlimit'] == 15
        assert c.configurations['listenqueuesize'] == 9
        assert c.configurations['socketbuffersize'] == 4096
        assert c.configurations['communicationsocketfile'] == '/tmp/sws.peerweb.it'
        assert c.configurations['errorlogfile'] == '/home/stefan/sws/log/error.log'
        assert c.configurations['accesslogfile'] == '/home/stefan/sws/log/access.log'
        assert c.configurations['errordocumentroot'] == '/home/stefan/sws/errordocs'
        assert c.configurations['communicationsocketfile'] == '/tmp/sws.peerweb.it'
        assert len(c.configurations['errordocument']) == 3
        assert c.configurations['errordocument'][403]['file'] == '403.html'
        assert c.configurations['errordocument'][404]['file'] == '404.html'
        assert c.configurations['errordocument'][500]['file'] == '500.html'


    def testConfigVHFiles(self):
        testfolder = self.CONFIG_FOLDER + '/t52'
```

configtestcase.py: PyUnit test cases for testing configuration file parser

```
        c = config.SwsConfiguration(testfolder)
        assert c.parseFile()[0] == True
        assert len(c.virtualHosts) == 2
        vH1 = testfolder + '/sites-enabled/127.0.0.1.conf'
        vH2 = testfolder + '/sites-enabled/watten.conf'
        assert c.virtualHosts.keys()[0] == vH1
        assert c.virtualHosts.keys()[1] == vH2
        # test VH1
        assert c.virtualHosts[vH1]['serveradmin'] == 'stefan@127.0.0.1'
        assert c.virtualHosts[vH1]['servername'] == '127.0.0.1'
        assert len(c.virtualHosts[vH1]['serveralias']) == 3
        assert c.virtualHosts[vH1]['serveralias'][0] == 'www.watten.org'
        assert c.virtualHosts[vH1]['serveralias'][1] == 'www.wattn.org'
        assert c.virtualHosts[vH1]['serveralias'][2] == 'wattn.org'
        assert c.virtualHosts[vH1]['documentroot'] == '/home/stefan/sws'
        assert len(c.virtualHosts[vH1]['directory']['/']['directoryindex']) == 3
        assert c.virtualHosts[vH1]['directory']['/']['directoryindex'][0] == 'index.html'
        assert c.virtualHosts[vH1]['directory']['/']['directoryindex'][1] == 'index.htm'
        assert c.virtualHosts[vH1]['directory']['/']['directoryindex'][2] == 'index2.html'
        assert c.virtualHosts[vH1]['errorlogfile'] == '/home/stefan/sws/log/a_error.log'
        assert c.virtualHosts[vH1]['accesslogfile'] == '/home/stefan/sws/log/a_access.log'
        assert c.virtualHosts[vH1]['errordocumentroot'] == '/tmp'
        assert c.virtualHosts[vH1]['errordocument'][404] == 'tmp404.html'
        assert c.virtualHosts[vH1]['errordocument'][500] == '500.html'
        assert len(c.virtualHosts[vH1]['directory']) == 4
        assert len(c.virtualHosts[vH1]['directory']['/']['cgihandler']) == 1
        assert c.virtualHosts[vH1]['directory']['/']['cgihandler'][0]['extension'] == '.asp'
        assert len(c.virtualHosts[vH1]['directory']) == 4
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin/sh']['cgihandler'][2]['extension'] == '.sh3'
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin/sh']['cgihandler'][2]['executor'] == '/bin/bash'
        assert len(c.virtualHosts[vH1]['directory']['/']['addtype']) == 2
        assert c.virtualHosts[vH1]['directory']['/']['addtype']['.css'] == 'text/css'
        assert c.virtualHosts[vH1]['directory']['/']['addtype']['.html'] == 'text/html'
        assert len(c.virtualHosts[vH1]['extfilterdefine']) == 2
        assert c.virtualHosts[vH1]['extfilterdefine']['test1'][0] == '/bin/test1'
        assert c.virtualHosts[vH1]['extfilterdefine']['test1'][1] == 'param'
        assert c.virtualHosts[vH1]['extfilterdefine']['test2'][0] == '/bin/test2'
        assert len(c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['setoutputfilter']) == 3
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['setoutputfilter'][0] == 'test1'
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['setoutputfilter'][1] == 'test2'
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['setoutputfilter'][2] == 'test1'
        assert len(c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['addheader']) == 2
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['addheader']['content-encoding'] == 'none'
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['addheader']['content-type'] == 'text/html'
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['stopinheritance']['directoryindex'] == True
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['stopinheritance']['cgihandler'] == True
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['stopinheritance']['setoutputfilter'] == True
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['stopinheritance']['addheader'] == True
        assert c.virtualHosts[vH1]['directory']['/docs/cgi-bin']['stopinheritance']['addtype'] == True

        # test VH2
        assert c.virtualHosts[vH2]['serveradmin'] == ''
        assert c.virtualHosts[vH2]['servername'] == 'watten.org'
        assert len(c.virtualHosts[vH2]['serveralias']) == 0
        assert c.virtualHosts[vH2]['documentroot'] == '/home/stefan/sws/docs'
        assert len(c.virtualHosts[vH2]['directory']) == 1
        assert len(c.virtualHosts[vH2]['directory']['/']['directoryindex']) == 0
        assert len(c.virtualHosts[vH2]['directory']['/']['cgihandler']) == 0
        assert c.virtualHosts[vH2]['errorlogfile'] == c.configurations['errorlogfile']
        assert c.virtualHosts[vH2]['accesslogfile'] == c.configurations['accesslogfile']
        assert c.virtualHosts[vH2]['errordocumentroot'] == c.configurations['errordocumentroot']
        assert c.defaultVirtualHost == vH2

if __name__ == "__main__":
    unittest.main()
```

List of Figures

2.1 Request processing of Apache 8

3.1 Example of a configuration with two Output Filters 14
3.2 Three example log entries of an access log file 18
3.3 Three example log entries of an error log file 18
3.4 Request processing with Privilege Separation 19
3.5 UML 2.2 Component Diagram 20
3.6 Creation of the root process 21
3.7 UML Class Diagram 24

4.1 Achieving asynchronous communication using *fork* 29

A.1 UML Class Diagram - Part 1 49
A.2 UML Class Diagram - Part 2 50
A.3 Browser establishes TCP connection 51
A.4 Listener establishes connection to root process 51
A.5 Connections established 52
A.6 CGI script execution in two different threads 52
A.7 Unit tests, performed using *PyUnit* 53
A.8 Webalizer 53
A.9 Gantt Chart 54

List of Tables

5.1 Benchmark results 42
5.2 Tested browsers 43

List of Program Listings

D.1 sws: start, stop, restart script of the server daemon 61
D.2 webserver.py: contains all three types of processes 62
D.3 httprequest.py: HTTP/CGI parser and request handler 66
D.4 config.py: configuration file parser 78
D.5 daemon.py: realises the daemon functionality (open source, public domain) 84
D.6 main.py: PyUnit main class for running all unit tests 86
D.7 servertestcase.py: PyUnit test cases for testing web-server functionalities . 87
D.8 configtestcase.py: PyUnit test cases for testing configuration file parser . . 91