UNIVERSITY OF GRONINGEN
Faculty of Mathematics and Natural Sciences
Department of Computing Science
Software Patterns

Apache HTTP Server

Pattern-based recovery & evaluation

GROUP 2
Buck, Werner
Heijkens, Jan-Alexander
Ioakeimidis, Spyros
Manteuffel, Christian
Katsantonis, Fotis
Hanjalic, Avdo

0.6
Groningen, February 12, 2012

Authors

Name	Acronym	E-Mail
Ioakeimidis, Spyros	si	spyrosikmd@gmail.com
Manteuffel, Christian	cm	cm@notagain.de
Katsantonis, Fotis	fk	fotiskatsantonis@gmail.com
Hanjalic, Avdo	ah	a.hanjalic@student.rug.nl
Buck, Werner	bk	wernerbuck@buck@gmail.com
Heijkens,Jan-Alexander	jh	jaheijkens@gmail.com

Revision History

Version	Date	Description
0.1r	19/12/11	Set up structure based on document template
	21/12/11	Cover image
0.2r	07/01/12	Introduction
		System Context
		Stakeholders and Concerns
		Key drivers
		Document candidate patterns
		Draft on Patterns documentation
0.3r	16/01/12	Software Architecture
		Elaborated patterns documentation
		Patterns diagrams
0.4r	23/01/12	Evaluation
0.5r	06/02/12	Recommendations
		Overall improvement
0.6	12/02/12	Conclusion
		Added Threadpool pattern
		Overall improvement

Legend: r = reviewed

Contents

List of Figures

List of Tables

1 Introduction

The Apache HTTP Server, hereinafter Apache, is an open-source server that aims at creating a robust, commercial-grade, featureful and freely-available web server. The Apache server is one of the most used web servers in the Internet.

This report presents the results of a project, which aims recover the architecture of the open-source Apache HTTP Server by identifying software patterns and to perform an evaluation of the architecture based on the identified patterns.

The report starts with a brief description of the system context and the environment of the Apache server including its history, the project owner and architectural relevant business parameters (cf. Section 2). Subsequently, the most important stakeholders and their concerns are presented (cf. Section 3). This information has been used to deduce the Key Drivers of the Apache Project, which are presented in Section 3.3. After this the recovered software architecture is illustrated by describing the decomposition of the system into components and connectors (cf. Section 4) and the actual usage of the identified patterns (cf. Section 5). Afterwards, the results of the architecture evaluation are presented (cf. Section 6). Therefore, Force-Resolution Maps and Quality-Attribute Pattern Matrices have been used. In Section 7, the gathered information about the architecture and evaluation is used to present recommendations and improvements for the Apache project.
Finally, the report concludes with a retrospection of the project itself and the used methodologies.

During this project, the IDAPO[1] process has been used to recover the architecture [4]. The PBAR[2] approach has been used to perform the evaluation [5].

The IDAPO process consists of twelve steps. Step 1 corresponds to Sections 2 and 3, in which the type and domain of Apache is identified. Section 4 is related to Steps 2 to 5 where used technologies and candidate patterns are examined.
Steps 6 to 12 is an iterative process, which results in the documentation of used patterns. The results are document in Section 5. During these steps, documentation, source code and the components and connectors have been studied. Furthermore, the identification and validation of patterns and variants has been realized.
Along with IDAPO the PBAR process determines the high-level process of pattern-based recovery and the evaluation of the architecture. The findings of this evaluation are documented in Section 6. An overview of the process mapping can be found in Appendix A2.

The project and the resulting report are conducted during the course Software Patterns at the University of Groningen.

[1]Identifying Architectural Patterns in Open Source Software
[2]Pattern-Based Architecture Reviews

2 System Context

In this Section the the Apache project is presented. In Section 2.1 the Apache HTTP server is discussed and in 2.2 the foundation behind this open source project, the Apache Software Foundation, is described.

2.1 Apache HTTP Server

The Apache HTTP server is a web server for Windows, Mac, Linux or other UNIX-based operating systems. The Apache server exists since 1995 as a patched version of the NCSA HTTP-server[3]. Some say that this is the reason for the name of the Apache server, 'a patchy server', however this is incorrect[4]. Apache stands for the native American Indian tribe of the Apache. This tribe was known for their superior skills in warfare strategy and inexhaustible endurance. The word "Apache" comes from the Yuma word for "fighting-men".

The Apache web server is an open-source project. It was designed by a group of webmasters, which wanted to distribute the patches made to the NCSA HTTP-server after the development stalled because the main developer quit. Eight core developers formed the foundation of the original Apache Group. Initially, the Apache server was the only open-source alternative to the Netscape web server. According to the Netcraft Web Server Survey in January 2012 [1] across the millions busiest sites 378,267,399 servers were running Apache. This means a market share of 64.91% (cf. Figure 1).

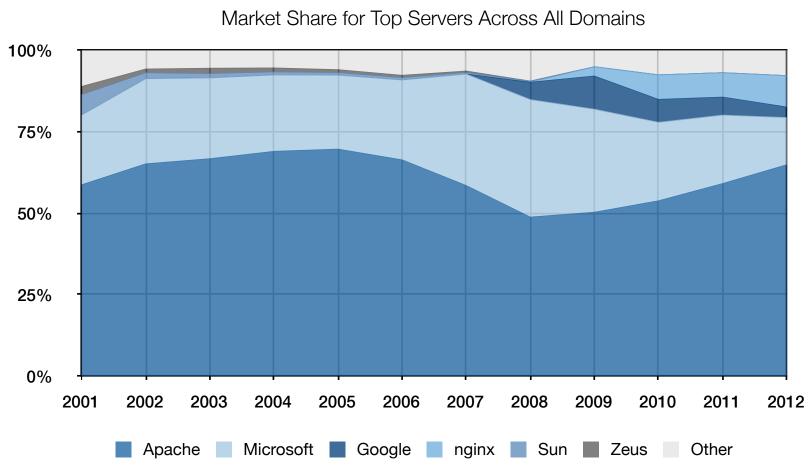

Figure 1: Market Share since 2001. Deduced from [1]

By making the Apache server project open-source the Apache Software Foundation believes that those who benefit from this software by using it, often contribute back to it by providing feature enhancements, bug fixes, and support for others in public lists and newsgroups. By making it free the contributions are higher, since typically thousands of independent programmers are testing and fixing bugs. This makes the product more stable and the penetration in the market improves.

It is possible for commercial companies to use the Apache server as a base for custom adjustments and attributions. Those adjustments can be incorporated easily, due to the modular architecture of Apache, which has been improved since version 2.0. Examples of frequently used modules are SSL, PHP or Proxy. This influenced the market penetration tremendously.

The Apache server and its modules are mainly written in the C programming language. However, it is possible to develop third-party modules in PERL. Apache uses the Apache Portable Runtime library and uses dynamic linking technology like Dynamic Shared Object.

[3] http://httpd.apache.org/ABOUT_APACHE.html [Online accessed; 17.01.2012]
[4] http://wiki.apache.org/httpd/FAQ#Why_the_name_.22Apache.22.3F [Online accessed; 17.01.2012]

Version 1.0 of Apache has been released in 1995. At the moment of writing, the latest version is 2.2.21, released on the 13.09.2011[5]. In this report version 2.2 is examined.

2.2 Apache Software Foundation

After the establishment of the Apache Group in 1996, some members of the group founded the Apache Software Foundation (ASF) in 1999. The ASF aims to provide organizational, legal, and financial support for the Apache HTTP Server and other Apache Foundation projects. Its goal is to make sure that Apache projects continue to exist beyond the participation of individual volunteers. The ASF is a meritocracy, which means that only developers who are in general actively contributing are chosen to lead the development of other projects. Besides the Apache server the ASF offers several famous projects, as illustrated in Figure 2. For instance, the build tools Apache Ant and Apache Maven, the Search Server Apache Solr or the content repository Apache Jackrabbit.

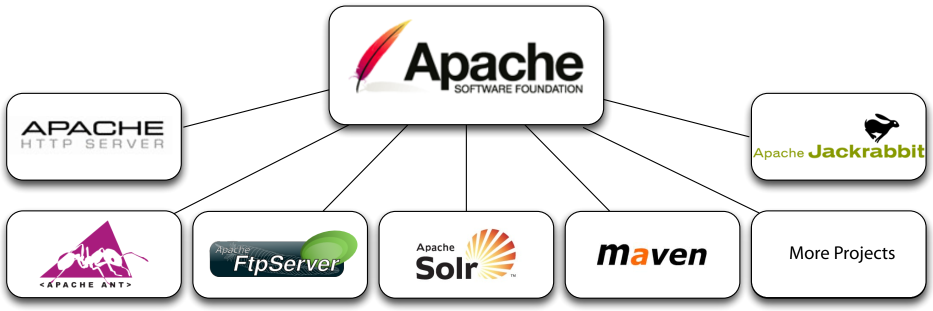

Figure 2: The Apache Software Foundation and some of their well-known projects.

The ASF is funded by a sponsorship program. This program exists of four grades with donations ranging from 5000 USD till 100.000 USD.[6] The ASF is a non-profit organization but with over 70 projects money is needed to pay for the everyday business operating expenses like office expenses, legal and accounting, marketing, public relations and technical costs as costs for the provided servers and hardware. With this sponsorship program and small donations from users, it is possible to fund their expenses and to deliver the Apache Server project open-source. The ASF is settled in Delaware in the United States of America.

2.3 Community

The Apache project relies on a community of volunteers. About 80 core developers are involved in the project and their responsibilities range from coding to documentation. In fact, the developers are listed as individuals but all work in some way in the IT industry. Companies may have wanted to use Apache and decided to let one of their workers work on a requested module.
There plenty of ways to communicate with the Apache developers. In general, the primary discussions and communications is done via mailings lists. But there are also yearly conferences, called "Apachecon". The conference provides a forum for collaboration and debate, and gives an opportunity for them to get to know each other.

The easiest and fastest way to get in contact with the developers is probably via IRC. The IRC has also been used in this project, to clarify the purpose of the core component in the Apache server, as well as to confirm the Key Drivers. The complete transcript can be found in Appendix A1.

- IRC #httpd and #httpd-dev on freenode.net
- Mailing Lists `http://httpd.apache.org/lists.html`
- List of Contributors `http://httpd.apache.org/contributors/`

[5]`https://httpd.apache.org/` [Online accessed; 17.01.2012]
[6]`http://www.Apache.org/foundation/sponsorship.html` [Online accessed; 17.01.2012]

3 Stakeholder and Concerns

This section list the stakeholders and their concerns for the Apache server. After that, the key drivers for the project are presented. The IEEE Std 9126-1 Quality Model was used in the definition of the quality attributes [3].

3.1 Stakeholders

Unless stated otherwise, the stakeholder and their concerns are assumed based on the existing documentation or the experience of the project members. The assumptions were made after thorough consideration of the concerns. For each stakeholder the name, formal identifier and priority are given, followed by an elaboration on the stakeholder's concerns. An overview of the Stakeholders and their concerns is illustrated in Table 5 in Appendix A3.

System Administrators (SH-01, High)

The system administrators are responsible for installation, configuration and maintenance the Apache HTTP Web server instances.

Source

The administrators are mentioned in the documentation of Apache. For example, the documentation contains a guide for system administrators to optimize the performance of the Apache server[7].

Priority justification

The system administrators opinion has a high weight on the server being chosen. If it cumbersome to setup and configure the Web server or the performance is bad, another competitors server will be favored.

Concerns

Stability	It is expected that the Apache HTTP Web Server is available for installation on a large number of platforms and that updates are provided on a regular basis for every platform.
Installability	System administrators expect that the Apache server can be installed in a specified environment.
Co-existence	A web-server often shares the resources of a server with other applications like databases. Hence, the Apache should not use resources mutual exclusive
Resource utilization	A server has limited resources, which has to be used in an efficient way. Purchasing additional server is expensive.
Security	The system administrator expects the security and privacy of their infrastructure. In general, it should not be possible to compromise the server, since this could have severe effects on a company's or organization's reputation.
Reliability	The system administrator expects the configured Apache HTTP Web server to act reliable in a way that it is predictable, i.e a robust memory management. The server should be highly available.
Adaptability	The system administrator expects the configuration of the web server to be easily configurable for his or her goals. This includes the ability to add functionality by enabling or disabling modules, or sites.

[7]http://httpd.Apache.org/docs/2.0/misc/perf-tuning.html [Online accessed; 17.01.2012]

Apache HTTP Server Project Management Committee (SH-02, High)

The group of volunteers who are responsible for managing the Apache HTTP Server Project. This includes deciding what is distributed as product of the Apache HTTP Server Project, maintaining the Project's shared resources, speaking on behalf of the Project, resolving license disputes regarding Apache products, nominating new PMC members or committers, and establishing guidelines.

Source

This group is derived from the Apache documentation[6].

Priority justification

The Apache HTTP Server Project Management Committee has the decision-making authority regarding the Apache web server.

Concerns

Stability	The Apache should avoid unexpected effects from modifications of the software, which has been mandated by the Management Committee.
Changeability	Is is important that the decisions of the PMC can be easily implemented in the Apache server.

Website Visitors (SH-03, High)

The group of visitors that request or rely upon a web page being served. Web-browsers expect backwards compatibility with previous HTTP protocols.

Source

As stated in Section 2, the Apache server is widely used, which indicates that many visitors, used the Apache in order to request websites. As mentioned in the FAQ, the visitors contribute to the community by testing the web server[8].

Priority justification

The website visitors are the end users of the Apache, hence they have a high priority If a server is not able to serve a webpage fast, visitors will not visit the page again. They are the final judges of whether the service is acceptable or not.

Concerns

Time-behavior	Visitors expect a web page to be shown in a reasonable amount of time.
Reliability	Visitors expect the system to work as intended.
Security	Visitors expect the Apache HTTP Web Server to be secure.

[8]http://wiki.Apache.org/httpd/FAQ#How_thoroughly_tested_is_Apache_httpd.3F [Online accessed; 17.01.2012]

Web-browser developers (SH-04, High)

The developers that write the collective group of software that relies on the HTTP server to serve the files needed to interpret and display a web page or to just fetch a web page.

Source

This group has been assumed to be stakeholder because web-browser are intermediates between visitors (SH-01) and the Apache server.

Priority justification

The web browser are the means of the end users for visiting pages. That is why this group has a high priority. Apache needs to make sure that the latest HTTP protocol and security standards are supported.

Concerns

Stability	Web-browsers expect the Apache HTTP server to always adhere to proper HTTP protocols. In other words, to ensure that Apache Web server does not drastically change over the course of time. Web-browsers expect backwards compatibility with previous HTTP protocols.
Security	Web-browsers expect the HTTP server to be compliant to security standards and protocols like SSL.

Website Developers (SH-05, Medium)

Website developers are the group of people concerned with the development of a website or service that needs to be served via HTTP.

Source

Website developers are the stakeholders that are deduced from the fact that Apache is used because of the need for web servers.

Priority justification

For website developers it is important that websites still work after a server update. Because most Apache servers are installed with the intention of serving a website. This stakeholder has a medium priority.

Concerns

Time-behavior	The website developer expects web pages to be served in an appropriate amount of time in relation to the complexity of their code.
Security	The website developer expects security features enabled in their code to be working on the HTTP server. No one should be able to change the website on the server, without permission.
Stability	The website developer expects their code to be working stable as with the version of Apache HTTP Server used during development and in future software releases.
Portability	Websites are often tested and deployed locally and later put on a productive environment. Hence, the Apache needs to be available different environments including desktop platforms.

Apache HTTP Server Committers (SH-06, Medium)

As described on the website, the Server Committers are "the group of volunteers who are responsible for the technical aspects of the Apache HTTP Server Project. This group has write access to the appropriate source repositories and these volunteers may cast binding votes on any technical discussion[6]".

Source
This group is derived from the Apache documentation[6].

Priority justification
The server commiters have a medium priority. They do not have a high priority like the website visitors for example because they are the ones providing the server, not the ones judging it, as is the case with the website visitors.

Concerns

Changeability	The commiters generally want the code to be easy maintainable to ensure that code fixes and additions can be realized with minor effort.
Security	The commiters are ultimately responsible for any security oversight in their code, they benefit from a strong set of rules and restrictions to ensure project integrity.

ASF Board of Directors (SH-07, Low)

The Board of Directors of The Apache Software Foundation is responsible for management and oversight of the business and affairs of the corporation in accordance with the bylaws. This includes management of the corporate assets (funds, intellectual property, trademarks, and support equipment) and allocation of corporate resources to projects. However, technical decision-making authority, regarding the content and direction of the Apache projects, is assigned to each respective project management committee.

Source
This group is derived from the Apache documentation[9].

Priority justification
The board of directors has a low priority based on the fact that the Apache web server is just one of the 70 projects from the Apache Foundation. They have influence on the project regarding allocating of resources but the decision-making authority is assigned to the Apache HTTP Server project management committee (SH-02).

Concerns

Changeability	The board wants to keep the HTTPD manageable. An easy to change system encourages new developers to contribute code in an open-source environment.
Portability	Releasing the Apache server for many platforms increases the market share and makes it easier to raise funds.

[9] http://www.apache.org/foundation/board/ [Online accessed, 17.01.2012]

ASF Security Team (SH-08, Low)

The Apache Security Team provides help and advice on security issues to Apache projects. The Security Team also coordinates the handling of security vulnerabilities.

Source

This group is derived from the Apache documentation[10].

Priority justification

The ASF Security Team has a low priority. They are not assigned to one project but work as an overall team for all the projects. They have influence but are not working on the project on a daily basis.

Concerns

Security	The ASF Security Team is concerned that the Apache server protects information and data of the end-users.
Analyzability	The Security Team mandates a high analyzability in order to quickly diagnose deficiencies or causes of failures.

Apache Developers (Contributors) (SH-09, Low)

Volunteers who contribute any time, code, documentation or resources to the Apache Project. These volunteers are different from the committers SH-06, as they do not have write access to the repository. They contribute anything they want on a un-official basis.

Source

These stakeholders are derived from the Apache documentation[6].

Priority justification

This group has a low priority for the same reasons as the server commiters SH-06, but they also do not have write access to the repository, that is why why a low priority has been assigned.

Concerns

The concerns of this group are covered by the server committers (SH-06) since they do not have the right to make any commits. The server committers review their code and decide on whether the change will be merged into Apache or not.

3.2 Negative Stakeholders

In this paragraph the stakeholders are mentioned that have an adverse effect on the project.

Hackers (NSH-01, High)

This group tries to exploit weaknesses in the Apache HTTP Web server in order to gain access to the rest of the infrastructure. Some hackers are more curious than aggressive and report these vulnerabilities as a sport to the ASF Security Team. But there are also aggressive hackers that use vulnerabilities for their own gain and keep the details to themselves.

Source

This group is not mentioned in the documentation. However, due to the tremendous market share, Apache servers are a likely target for hackers.

Priority justification

This group has a high priority since they are trying to compromise the server. Thus the development team and the community need to implement effective security measures.

[10]http://www.apache.org/security/ [Online accessed; 17.01.2012]

3.3 Key Drivers

In this Section the key drivers of the Apache HTTP server project are mentioned in order of priority. The first three key drivers can be found in the Apache website.

> The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows NT. The goal of this project is to provide a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards [11].

Efficiency is perceived as performance including resource utilization and time-behavior. The portability key driver was not found in the documentation but it is obvious that it is their concern since a wide range of operating systems is supported. This fact is undermined by the Stakeholder-Concern Matrix in Appendix A3. The Extensibility is not a quality that is mentioned by the quality model. However, it is explicitly stated in the documentation. Extensibility can be mapped to Changeability but also to Adaptability. While the first one considers source code changes during development time, the second one is focused on extension during compilation or runtime.

Priority was assigned by the authors in order to be able to justify trade-offs during the evaluation. There was no information about the priority of these quality attributes on the Apache documentation.

1. **Extensibility**
   Apache HTTP Server has to be able to provide a feasible and easy way of extending its abilities. It is released as an open-source project and therefore it must be easy for third parties to further develop it, adding extra functionality, according to their needs.
2. **Security**
   Applications which use the web server are confronted with the issue of a single point of entrance. The web server needs system resources for its operation and the use of it implies exposure of those resources and additional content to public through the internet. The server fulfills all the prerequisites for securing the content and everything that is built upon it. The server ensures the prevention of basic security threats related to the web. The fact that Apache has to run partially as root in order to listen on port 80, emphasizes the focus on security.
3. **Efficiency**
   Apache HTTP Server is intended to serve complex, large and highly requested websites. Each website relies on the server's ability to use its resources efficiently in order to provide a high performant service.
4. **Portability**
   The server is offering cross-platform functionality. Apache HTTP Server is meant to be used in different environments, with different standards, including operating systems.

[11] `http://tldp.org/HOWTO/Apache-Overview-HOWTO-2.html` [Online accessed; 17.01.2012]

4 Software architecture

In this Section the architecture of the Apache HTTP server, also known as Apache HTTPD, is discussed by means of a logical view and process view. First the building blocks are discussed, in order to explain the multi-processing capabilities. Finally, the handling of a service request and required module communication is presented. Figure 3 shows the high level architecture of the Apache.

4.1 Logical View

From a basic point of view the Apache server consists of two major components: the core and the modules. The core is the main process that is started first and handles the basic functionality of the server. The modules are parts of functionality that can be added to the Apache server on demand. This is depicted in Figure 3 and explained more elaborately in the following subsections.

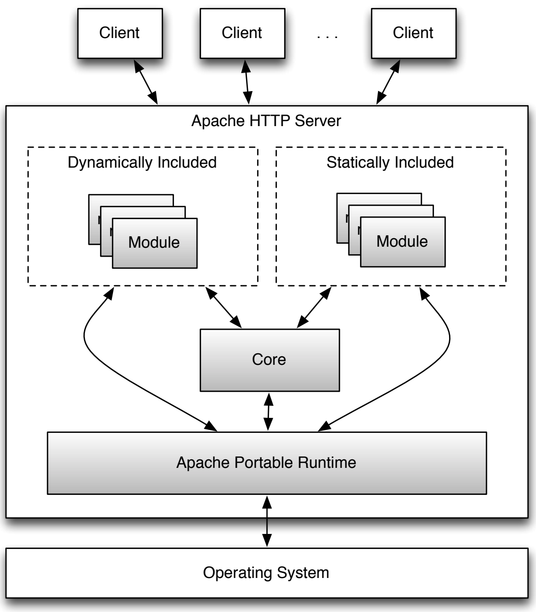

Figure 3: High Level Architecture of the Apache HTTP Server

4.1.1 Core

The core functionality can be summed up as:

1. **Configuration:** Before the core can load modules, it needs to know what modules to load. This is defined in the configuration and thus loading, parsing and handling configuration is a part of the core's functionality.
2. **Loading of modules:** The modules are loaded by the core. Each module represents a specific functionality like cgi, php or file-listings which can either be statically or dynamically loaded. The distinction between static and dynamic inclusion will be explained in the next section.
3. **Request processing:** Apache HTTP Project is a request processing server, and each request is passed through the core. Specifically, the core pipes the request through several filters and hooks, these filters and hooks are provided by the afore-mentioned modules and are hooked into the core for subsequent handling.
4. **Memory allocation:** The core provides an API for the modules. The modules can subsequently ask the core for more memory for processing.
5. **Start-up/Shut-down/Logging:** Maintaining the main loop, ensuring a clean exit with memory properly released and ensuring multiple levels of logging are also handled by the core.

As can be seen from the list the core is kept as minimal as possible, with most of Apache HTTP Project's functionality being implemented as modules.

Communication The communication between the core and modules is not on equal footing. The core can only call handlers for registered hooks and the modules can only communicate with the core or other modules through the Apache API. For example to read and modify a request/response record *request_rec* and to ask the core to allocate memory to the request or connection pools.

4.1.2 Modules

The Apache server follows a modular approach. It is able to extend or provide new functionality through modules. Modules can either be statically or dynamically included by the core. For static inclusion the module source code has to be added and compiled with the server. A dynamically included module adds functionality by being loaded as a shared library during start-up or restart of the server. The functionality to add modules dynamically in a POSIX environment is actually provided by a static module called *mod_so*. All but the very simplest of functionality was moved from the core to modules after version 2.0.
The dynamic modules are called Dynamic Shared Objects (DSO). These DSOs can either be compiled together with the server or they may be compiled individually and added later. The advantage of DSOs is that it increases the flexibility of the server during runtime.
Flexibility is also provided to distribution vendors, they can bundle modules as extensions. This gives users the choice of installing whatever software they need. Examples of such modules are *mod_php5* and *mod_fastcgi*. The behavior of loading modules coincides with the architectural pattern called Plugin, which will be described in section 5.2.2.

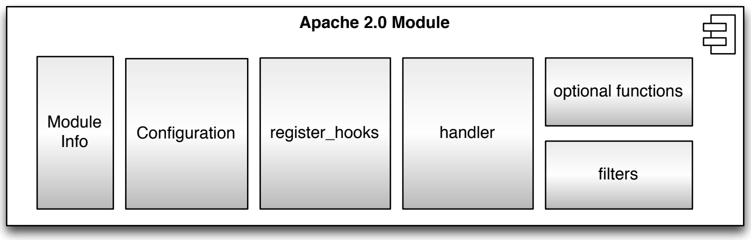

Figure 4: Apache HTTPD 2.x module structure. Deduced from [2]
.

Figure 4 illustrates the structure of a module. Each module has *Module Info*, that provides information about the handlers that it provides. It also contains what configuration directives the module can process. The *Module Info* is required by the core, in order to properly initialize and register modules.
In order to trigger modules to perform their tasks at the appropriate moment, hooks are introduced in Apache HTTPD. Modules can register handlers to hooks, if they have actions to perform for the specific hook. HTTPD provides hooks for:

1. Managing and processing configuration directives
2. State changes of the server (e.g. start-up, shut-down, restart)
3. Processing of incoming requests

There can be different kinds of handlers in modules:

- **Hook handlers**: A hook in the Apache server is an event where all the registered handlers are called.
- **Configuration handlers**: These are handlers that provide processing for module configuration.
- **Optional Functions**: These are the same as hooks, but the difference is that the core ignores any return value from an optional function. They are called regardless of error and are used for tasks that are not crucial to the request-response process.
- **Filters**: These are handlers for processing data of the request and the response. They are discussed in more detail in the process view.

4.1.3 Apache Portable Runtime

Since the goal of Apache is to be the most used web-server, it has to support many platforms. A common problem for porting software is that operating systems handle I/O, memory management, locks, networking, and dynamic linking differently.
For this reason, the Apache Portable Runtime (APR) was introduced in version 2.0. This enhances functionality to the Apache API and makes Apache the portable server that it is. The APR is an Indirection Layer (cf. Section 5.3.1), that provides wrappers for all operating system functions the core or any module needs.
The APR alongside with the Apache core can both be used to create any request processing server imaginable. However the APR can be bypassed by modules if needed. For instance, it is bypassed by the mpm_worker module, to increase performance by skipping the extra indirection layer.

4.1.4 Multi-processing

Service applications like Apache HTTPD have to utilize the full potential of the machine that executes them. Multi-processing is a technology broadly used to increase performance of applications: multiple CPU cores are utilized to perform tasks simultaneously, therefore the required execution time is reduced.

The Apache server supports a variety of Multi-Processing Modules (MPM), which have to be included during compile time[12]. Only one MPM can be used at a time. The most commonly used MPM since Apache 2.0 is mpm_worker. Because Apache HTTPD 1.x handled multi-processing by pre-forking processes, mpm_prefork is also included in version 2.x for backwards compatibility.

Worker
When the Apache HTTPD is compiled with the worker multi-processing module, the main executable will launch child processes, which in turn will span an amount of threads specified in the configuration file. In order to reduce delays for new connected clients, spare threads are maintained in a pool. The maximum number of threads in the pool is also defined in the configuration file. During operation, the server estimates the total amount of idle threads in all processes. If the amount exceeds the pre-defined minimum or maximum, then processes are killed or forked in order to regain the desired amount of idle threads.

Performance is gained by using the worker MPM, because threads can share data and module instances and can be kept alive for quick responses. However, while data and module sharing improves performance, this creates a single point of failure for each thread pool. If a module crashes, it will become unavailable for all threads within the same pool. Furthermore, as threads are started from the same process, they can share data and therefore can cause other threads to behave maliciously.

Prefork
The prefork multi-processing module will cause the Apache server to run separate processes instead of threads, for request processing. This MPM is typically used when threading needs to be avoided, e.g. for compatibility with non-thread-safe libraries. The main prefork process manages the amount of idle processes and attempts to maintain the amount of idle processes within some bounds, in order to keep the server's response time as low as possible.

This MPM will have a positive impact on the system's security: processes cannot read or modify each others data. In case a module crashes in some process it will not affect other processes, because all module instances are loaded individually.

The extra security comes at the expenses of a heavy RAM utilization. This will limit the amount of maximum concurrent processes and therefore the overall performance. System administrators will have to limit the amount of maximum processes in order to prevent RAM thrashing (swapping memory to disk), as it will render the server useless.

[12]http://httpd.apache.org/docs/2.3/mpm.html [Online accessed; 17.01.2012]

4.2 Process View

To get a better insight in the functionality of Apache HTTPD components, the interaction between these components is discussed in this section. The components altogether have only one target, i.e. handling requests, which is executing in a predefined sequence of steps.

4.2.1 Request Handling

When the first request from a client is received by the communication service of the operating system, it is forwarded to the MPM, which subsequently delegates the new created connection to the core. The core then initializes all necessary modules and binds the connection with a process. The following requests are forwarded directly to this process, as it now owns the connection.

The request is sent to a request processing agent. The agent then sends the request through a chain of filters, to eventually deliver it to the content handler. An example of a content handler for HTML is PHP. Once the content handler has generated output, the output is sent through the output filter chain. Section 5.1.1 elaborates the associated pattern, called Pipes and Filters.

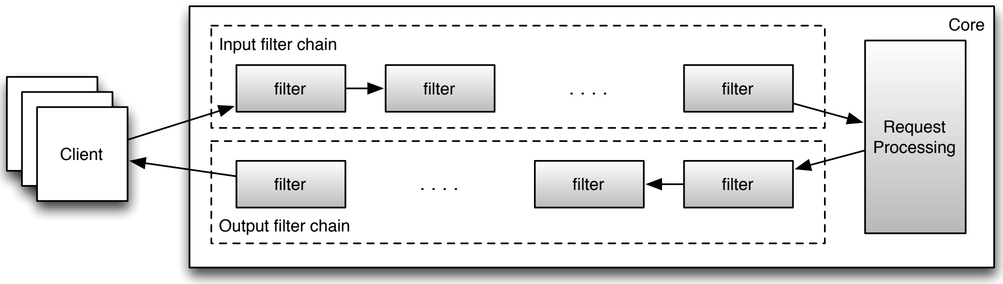

Figure 5: Apache Filters. Deduced from [2].

Figure 5 illustrates the basic sequence in which a request is processed, starting from its arrival, to the core and delivering the response. Messages are passed through a set of filters of which the order is pre-defined. Three different categories of filters can be distinguished:

1. **Connection and Network Filters** are used for handling low-level connection affairs, e.g. the SSL/TLS handshake between client and server for establishing an HTTPS connection.
2. **Protocol and Transcode Filters** handle the message protocol, e.g. HTTP.
3. **Resource and Content Set Filters** are responsible for generating the content that is returned to the client. PHP is an example of such a filter.

4.2.2 Data passing among filters

Filters, which are implemented in modules, need to communicate with each other. For performance, filters do not copy processed data to the next filter. Instead they share a reference to data containers, also known as buckets. Each request is split up into brigades, which in turn consist of buckets. A filter manipulates one bucket at a time and passes a reference of the bucket to the next filter. This process is described in detail in Section 5.1.2.

5 Pattern Documentation

In the following sections the patterns are described, which have been identified in the architecture of the Apache Webserver. The presented patterns are those, which are perceived to have a significant impact on the architecture of the Apache server and its qualities. The patterns are documented according to the template described in [7].
Additionally, each pattern description consists of a diagram that illustrates the collaborators, the pattern decomposition or the runtime behavior of the pattern. Furthermore, the template has been extended with a paragraph about traceability, which describes where and how the pattern has been identified within the architecture.

5.1 Core

The core is responsible for request processing, as mentioned in Section 4.1.1. Two patterns have been identified, which are related to the core component. These are Pipes and Filters (cf. Section 5.1.1) and Shared Repository (cf. Section 5.1.2).

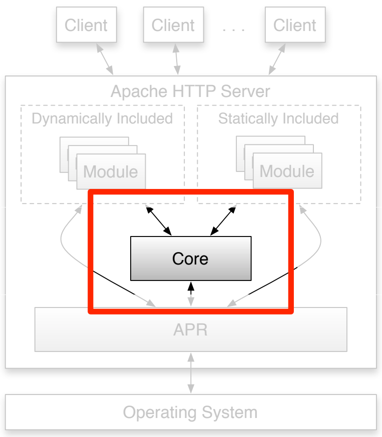

Figure 6: The Core component in the high-level architecture

5.1.1 Pipes and Filters

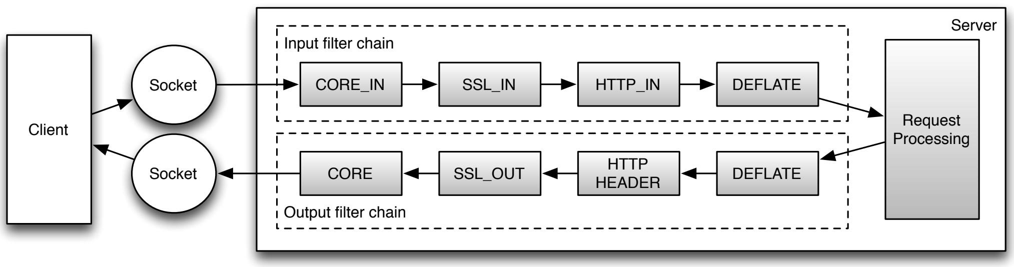

Figure 7: Input and output filter chains for the HTTP request and response processing

Traceability

This pattern is deduced from the Apache documentation[13]. Input as well as output filters are mentioned being responsible for processing the data received or sent by the server. Additional information is extracted from [2].

Source

The pattern used is a variant of the Pipes and Filters pattern, which is documented in [8] and in [9].

Issue

One of the main functionalities of the Apache HTTP Server is processing the data of HTTP requests and responses. During this process many operations need to be executed by modules related to this process. Processing requests and responses is a complex task, the realization of it by one monolithic component to encompass this functionality is inefficient.

Assumptions/Constraints

It is assumed that the operation of HTTP request and response processing can be split into subtasks.

Solution

Use the Pipes and Filters pattern to structure the process. The request and response process is split into multiple handlers, creating several sequential subtasks. The modules hook to these handlers, which constitute the Pipe and Filter chains. Each filter is a module and responsible for a particular operation.

Two filter chains are being used, the *input* and the *output* chain. The first one is used for processing the data of HTTP requests and the latter for processing the data of HTTP responses. The input filter chain is triggered from an agent called *Request Processing*. The output filter chain is triggered by the content handler. Between the filters is a constant flow of data related to HTTP requests and responses.

Operations, which are executed by both of the filter chains are related to network and connections, protocol, transcode, resource and content set. For example the SSL Module in the input filter chain provides secure HTTP communication known as HTTPS. Another operation, this time from the output filter chain, is the compression of a resource depending on its type.

The variant used is called Relaxed Pipes and Filters. This is due to the use of a Shared Repository (cf. Section 5.1.2) which provides filters a common datastore.

Rationale

By splitting up the operation of HTTP request and response processing in chains of Pipes and Filters, the complexity of such an important task is considerably decreased.

Implications

- A successful return of a response to a client depends on the fact that all subtasks have been executed successfully. In the case of an error in a module, the whole operation will be aborted.
- Flexibility is enhanced because the implementation of this pattern provides the possibility of alternative composition of filter chains.
- In addition the use of Pipes and Filters provides extensibility. Additional modules can be added to the initial process addressing different issues.
- Transferring data between filters may result in efficiency overheads or buffer overflows.

Related Patterns

- Shared Repository 5.1.2
- Interceptor 5.2.1

[13]http://httpd.apache.org/docs/2.0/filter.html [Online accessed; 17.01.2012]

5.1.2 Shared Repository

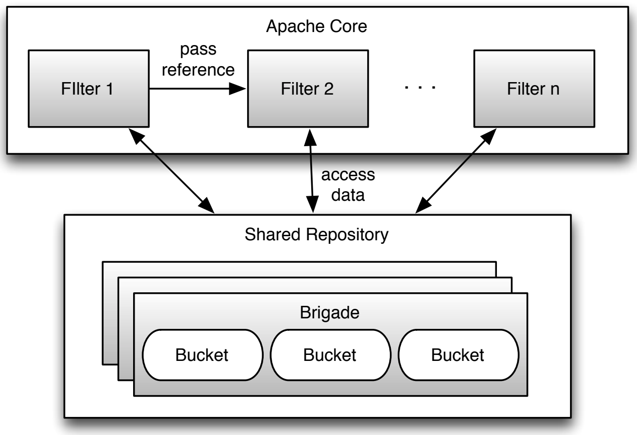

Figure 8: A shared data repository used by filters

Traceability

This pattern is used in combination with the Pipes and Filters Pattern, as described in [2]. In Apache source code documentation it is described as a complex data stream that can be passed through a layered IO system without unnecessary copying.

Source

The Shared Repository pattern is documented in [8] and in [9].

Issue

Each HTTP request or response is processed by multiple Filters. Therefore each filter must sequentially transfer all the data to the next filter's input. The same data is expected to be altered by every Filter. Transferring all the data between them is inefficient.

Assumptions/Constraints

Solution

The filter chain, which is responsible for the HTTP request and response processing, is using a Shared Repository. The repository consists of brigades, which in order consist of a number of buckets. Shared Repository is used for data sharing between filters and it is realized using RAM memory resources.

Filters work independently from each other by splitting the data into buckets and brigades. Data related to every HTTP request and response is split into several brigades. Instead of passing whole data packages to the next filter's input, filters pass references to that data, so the next one in the sequence is able to use the stored data.

For instance, one filter handles one bucket at a time and when finished, it passes the bucket to the next Filter (cf. Section 4.2.1). Still the order in which the filters hand on the data is kept intact. In this way, filters communicate directly with each other passing information where the data are located instead of transmitting the actual data. Filters passively access the Shared Repository.

Each process or thread that uses Pipes and Filters architecture for request handling, creates its own set of brigades and buckets. In this way a Shared Repository is used locally related to a thread or a process. At the end, everything that is not needed is destroyed.

Rationale

Shared Repository provides an efficient way of sharing data between the filters. Filters, as previously mentioned, pass references to the next filter instead of whole data packages. Hence, there is a loose dependency between the filters.

Implications

- With the use of a Shared Repository performance is greatly improved.
- Modifiability as well as maintainability are improved because the filters are mainly connected through the data store.
- The fact that only one filter can access a bucket at a time, negatively affects concurrency.
- The implementation of this pattern introduces extra complexity.

Related Patterns

- Pipes and Filters 5.1.1

5.2 Modules

The modules subsystem utilizes the Plugin pattern and the Interceptor pattern. Plugin provides the capability of loading modules during runtime (cf. Section 5.2.2). Interceptor implements the hooking mechanism, which is used by the modules (cf. Section 5.2.1).

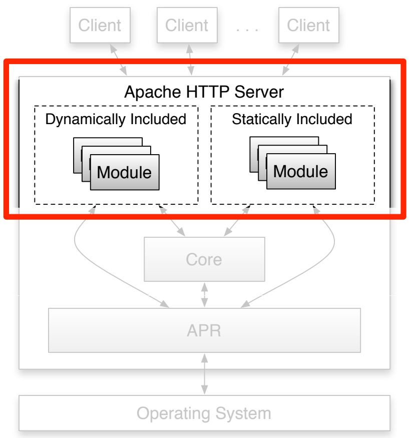

Figure 9: Relation of modules subsystem patterns with high level architecture

5.2.1 Interceptor

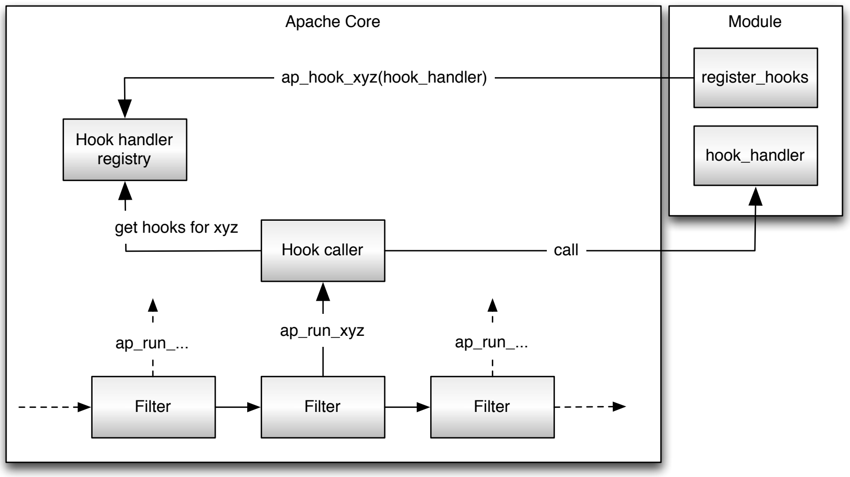

Figure 10: Intercepting a filter by registering for the hook xyz

Traceability

The Interceptor pattern has been deduced from the official documentation[14].

Source

The Interceptor pattern has been documented in [9].

Issue

The Apache HTTPD has a predefined sequence that describes how to process requests and generate responses. The server aims to be modular and thus the way it handles requests needs to be extensible for the future.

A mechanism is desired that allows to add functionality to the default request processing without the need to change the source code.

Assumptions/Constraints

Hooking points have to be defined extensively in advance, in order to allow a certain degree of flexibility in adding useful services.

Solution

Provide a hooking mechanism that allows to intercept the default request processing sequence by using the Interceptor pattern.

Rationale

The Apache HTTPD allows by default to intercept the configuration management process, the startup and shutdown process as well as receiving and processing of requests (cf. Section 4.1.2). Within the server a point that can be intercepted is called a hook, while the interceptor itself is called a hook_handler.

Multiple handlers can be registered at a particular hooking point. The owner of the hooking point decides whether to run all handlers or only the first successful.

Apache provides a central point to register hooks and dispatch registered hooks. Each module or filter is able to create its own hooking points that can be used by other modules.

Implications

- The Interceptor pattern increases modifiability and extensibility because existing process can be extended easily.

[14] https://httpd.apache.org/docs/2.0/developer/hooks.html [Online accessed; 17.01.2012]

- Security is problematic because the order in which handlers are executed cannot be easily traced and the execution of a particular handler or a combinations of handlers can have unforeseen implications during runtime.
- Similar to security, efficiency can be negatively affected if a lot of handlers have to be executed.

Related Patterns

- Plugin 5.2.2
- Pipes and Filters 5.1.1

5.2.2 Plugin

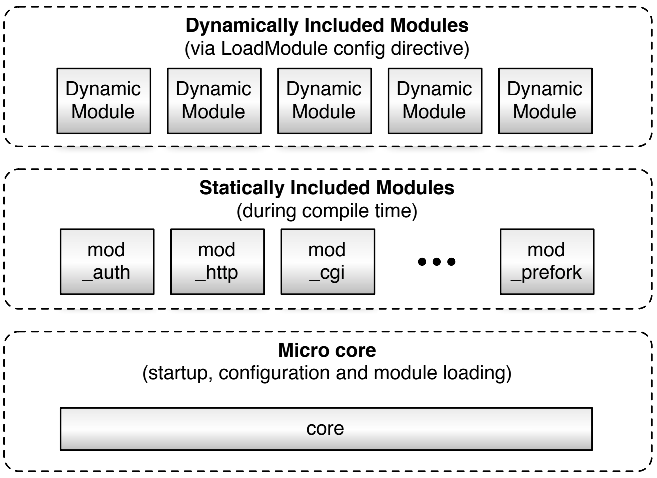

Figure 11: At startup the Apache core loads statically and dynamically included Modules

Traceability
The documentation of Apache 2.0 describes the usage of Dynamic Shared Objects (DSO). This technology is used to dynamically load modules, which indicates the presence of the Plugin pattern or a variant of it.

Source
The Plugin pattern is documented in [9].

Issue
The Apache server only provides a core functionality but the requirements for a web server are extensive. Thus a mechanism is required to add functionality in order to extend the basic functionality of Apache. Recompiling Apache is not feasible in a productive environment.

Assumptions/Constraints
The operating system has to support dynamic loading of code into the address space of a program.

Solution
The Plugin pattern resolves the problem of extending functionality at runtime and compile time by providing an interface and a central point to manage plugins. In this case plugins are called modules.

Rationale
The Apache server supports two types of modules. Either statically included or dynamically included modules. Statically included modules require a recompilation of HTTPD together with the module source code. In contrast, dynamic modules can be added during runtime via the LoadModule configuration directive, which can be added without shutting down the server. It is sufficient to initiate a reload of the configuration files.

The DSO technology allows to compile and include modules separately from the HTTPD binaries. Each module, whether it is static or dynamic, has to include a Module Info, which is essential for registration with the core. For instance, hook handlers are registered. In general, the Plugin pattern uses the Interceptor pattern (cf. Section 5.2.1) to extend the Apache by registering certain handlers.

Implications

- The Plugin pattern improves the extensibility of the system. In this case it is increased because it allows the administrator to load modules during runtime without the need to recompile the server.
- Adaptability is increased because a HTTPD instance can be adapted in various ways to address certain requirements. Portability is also affected positively because modules can be ported independently to a specific target platform.
- It slightly impedes performance due to the additional overhead.

Related Pattern

- Pipes and Filters 5.1.1
- Interceptor 5.2.1

5.3 Apache Portable Runtime

The APR increases the portability of the Apache server, by abstracting operating system functionality. It can be seen as an Indirection Layer pattern (cf. Section 5.3.1).

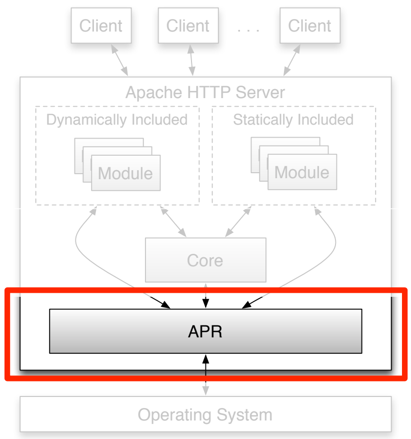

Figure 12: The APR in the high-level architecture

5.3.1 Indirection Layer

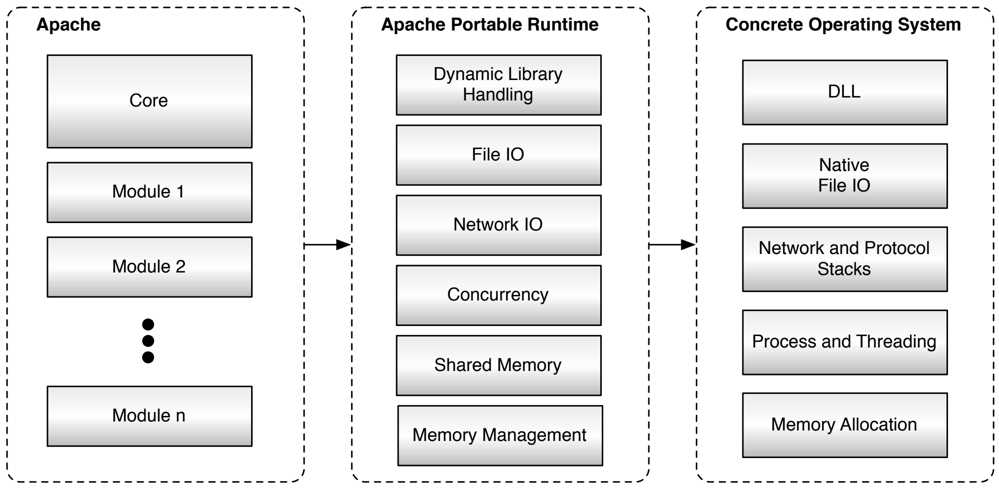

Figure 13: Apache Portable Runtime Indirection Layer

Traceability

The pattern has been deduced from the Apache source code documentation, in which the Apache Portable Run-time is described as a library designed to provide a common interface to low level routines across any platform.

Source

The used pattern is a relaxed version of the Indirection pattern, which is a variant of the Layers pattern[8] and has been described in [9].

Issue

Apache HTTPD is a multi-platform software. Amongst others, it has been ported to Linux, Windows, Mac OS X and NetWare. However, each operating system has its own API and characteristics when it comes to File I/O, Networking I/O or Memory Allocation. Furthermore, each OS provides a different set of functionality to deal with concurrency and multi-processing.

Porting the Apache HTTPD server to arbitrary operating systems involves writting platform-depended code. Identifying the specific platform-depended parts within the sources requires a lot of effort. Furthermore maintaining different versions of Apache can introduce code duplication and bugs.

Assumptions/Constraints

It is not compulsory to use the APR but it is strongly recommended.

Solution

Move the platform-dependent code into a central component and introduce a layer of indirection, in this case the Apache Portable Runtime (APR, cf. Section 4.1.3).

The APR wraps the platform-dependent API and provides a common API that is used by the core and modules. Amongst others, the APR provides an API for memory allocation, thread and process spawning, locking, file I/O and network I/O. Where the OS does not support a particular function, APR will provide an emulation.

Rationale

The introduction of the APR allows developers to rely on predictable behavior of code regardless of the target platform. The Indirection Layer provides a central point to maintain and evolve all platform depended code. This makes it easier to support different operating systems and target platforms.

However, in some cases it is more efficient to write platform-specific code and avoid the additional abstraction layer. For instance, the documentation[15] mentions the mapping of requests to either

[15]Sources: httpd-2.2.21/srclib/apr/docs/APRDesign.html

threads or processes as an example, in which platform specific code should be favored over APR. Hence, using the APR is advised but not mandatory in any case.

Implications

- Due to the encapsulation of platform-specific code, the usage of the Indirection Layer pattern, increases the portability of the system.
- However the APR introduces additional calls and indirections, which has implications to the efficiency of the Apache HTTPD server. In order to increase efficiency, a relaxed version of the Indirection Layer is used, which allows to bypass the APR if necessary.
- A security mechanism can be implemented easily within the APR because a central component exists (e.g. a pooled memory management), which reduces the probability of memory-leaks. Furthermore, the Indirection Layer reduces clones within the source code, which further improves security due to less bugs.

Related Patterns

The Indirection Layer does not actively collaborate with any other identified pattern, although there are patterns used within the APR like the Factory Method [10] in the crypto module[16].

5.4 Multiprocessing

The patterns described in this section are part of the multiprocessing subsystem. They are implemented by statically loaded modules. These patterns are used mainly during request processing. Since this is realized in the core, it is not possible to clearly assign these patterns to either the core or the modules component.

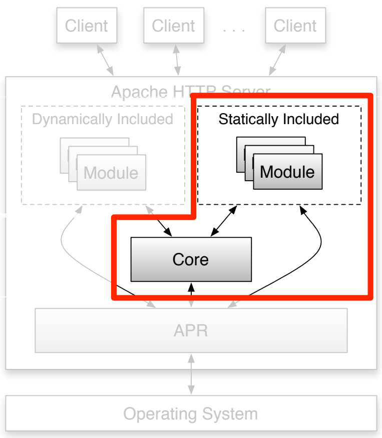

Figure 14: Multiprocessing related components

[16]https://apr.apache.org/docs/apr/trunk/apr__crypto_8h-source.html [Online accessed; 06.02.2012]

5.4.1 Leader/Followers

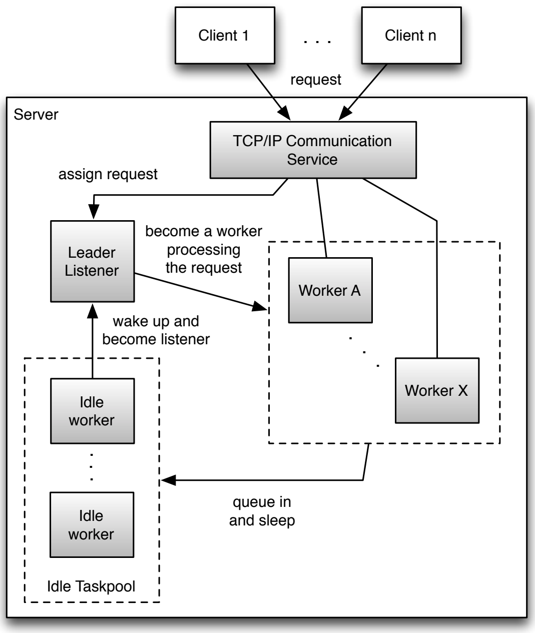

Figure 15: Listener, idle worker and worker tasks used for request processing

Traceability

Apache uses a selection of Multi-Processing Modules (MPMs) as mentioned in the Apache source code documentation[17]. As described in [2] this pattern is used in combination with these multi-process and/or multi-thread implementations (cf. Section 4.1.4).

Source

This pattern is a variant of the Leader/Followers pattern, which has been described in [11]

Issue

The Apache server implements the multi-processing technology by using processes or threads, in order to be able to handle multiple requests at the same time. Due to multiple type of requests that arrive simultaneously, performance is impeded.

It is inefficient to associate a task with a request mainly due to the limitations of the underlying operating system and network platform. Therefore, determining efficient demultiplexing associations between tasks and requests is needed.

Multiple tasks that accept requests from the TCP communication service must coordinate in order to avoid race conditions. Race conditions can occur when multiple tasks try to access or modify requests simultaneously.

Assumptions/Constraints

The task pool must be created at the server start. The amount of tasks in the pool should be large enough to ensure quick server response. The available machine resources limit the maximum number of threads that can be alive at the same time.

Solution

In the Leader/Followers pattern, roles are assigned to tasks in a pool. Tasks can either be threads or processes. The leader is the listener. Only one task can be granted the right to wait for connection requests. When a request with a connection arrives at the server it is coupled with the listener.

[17] http://httpd.apache.org/docs/2.0/mpm.html [Online accessed; 17.01.2012]

At this time an idle worker, a follower, is becoming the new listener and is waiting for a new connection. The previous leader then becomes a worker and it starts processing the request. When a worker finishes the request processing, it becomes idle again and is ensued into the pool of followers.

Rationale

Applying this pattern has the advantage that incoming requests are treated immediately by the listener task. There is no need of creating a new task for this purpose. Also there is no need of passing information between tasks because tasks simply change roles keeping the information and using it.

Implications

- The Leader/Followers pattern improves performance and ensures efficiency of the server, because requests are served almost instantly.
- Complexity is introduced in the implementation.

Related Patterns

- Secure Pre-Forking 5.4.2
- Threadpool 5.4.3

5.4.2 Secure Pre-Forking

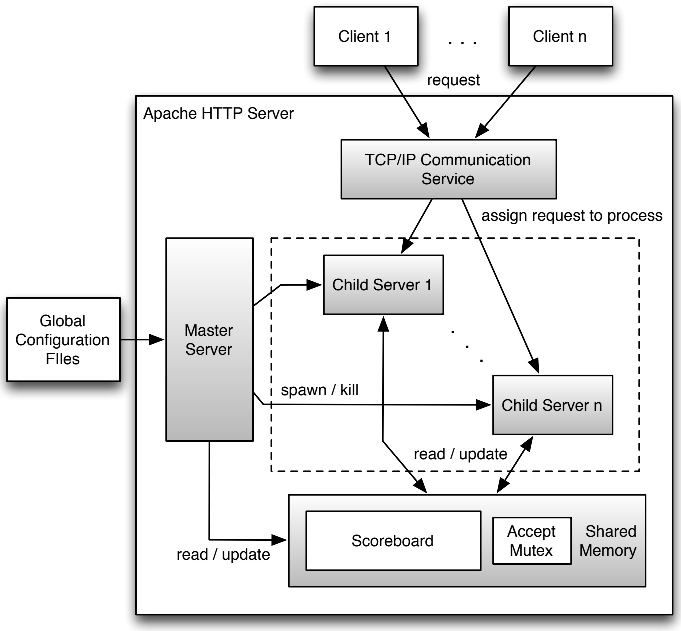

Figure 16: Pre-forking child server processes for serving HTTP requests and process accounting

Traceability

This pattern is deduced from the Apache source code documentation[18] and is described as a module, which implements a non-threaded, pre-forking server responsible for the request handling. Additional information is extracted from [2].

Source

The Secure Pre-forking pattern is described in [12].

Issue

Web servers act as gatekeepers waiting for incoming requests. By using the forking technique to create child processes upon a request arrival, it impedes performance due to heavy memory consumption. Moreover child processes created by the master server in order to serve incoming requests, are residing in a resource pool. After the completion of serving a request, these processes remain alive until the server is shut down. This can encompass several security issues.

Assumptions/Constraints

It is assumed that cloned processes do not conflict.

Solution

Pre-forking technique is used to spawn child processes before any request arrives in the server, so they can be instantly served. The amount of these processes to be spawned is configured through an external configuration file.

The number of processes that remain idle is controlled by the master server in order to ensure that a request can always be served instantly. A new connection is forwarded to the child process holding the accept mutex, which is released by the child process once the request is received. Now another process can lock the mutex in order to retrieve the next request. After successfully serving the request the child process returns to the resource pool.

Limiting the lifetime of daemon processes along with the pre-forking technique, increases the security of the process. Multiple parameters are considered to identify processes that need to be killed. The identified processes are killed and new processes are forked taking their place.

The Apache server has various configuration parameters that are used during server startup and process management. Data structures are maintained for process accounting by the Master Server. These structures are stored in a shared memory with the form of a table called scoreboard. Based on these, processes are killed and replaced after their limited lifetime.

Rationale

It is inefficient to wait for incoming requests and spawn new processes. A lazy forking by the master server will mean that during this period, it will not be able to serve more requests. However this could result in a resource pool of unused processes.

Processes should not be reused infinitely. This can lead to vulnerable systems with several security issues, due to memory leaks for example. This is the main reason for using a limited lifetime parameter along with other parameters in order to monitor child server processes and evict them.

Implications

- Secure Pre-forking enhances the security of the system by limiting the lifetime of the daemon processes, thus reducing the effect of a compromised process.
- Performance is increased since requests are served almost instantly by the processes. However evicting and spawning processes introduces additional overhead. Therefore the performance increase is somehow limited, but still present.
- Secure Pre-forking leads to a more complex architecture. This could mean less portability, as well as more faults.
- Availability may be negatively influenced. If many processes need to be killed, then many requests may return unserved.

Related Patterns

- Leader/Followers 5.4.1
- Threadpool 5.4.3

[18]http://httpd.apache.org/docs/2.0/mod/prefork.html [Online accessed; 17.01.2012]

5.4.3 Threadpool

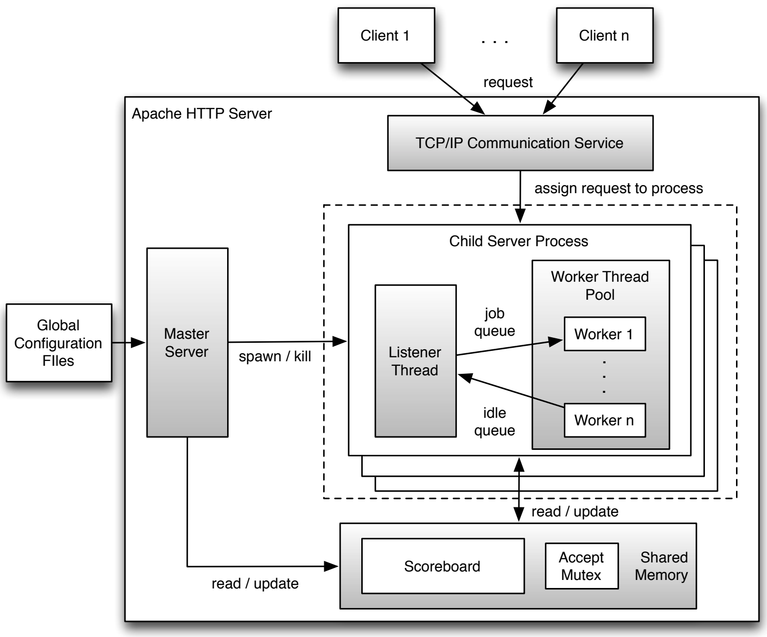

Figure 17: Child server processes with thread pools

Traceability

This pattern is deduced from the Apache source code documentation[19] and is described as a module, which implements a multi-threaded server responsible for the request handling. Additional information is extracted from [2].

Source

The Worker Thread pattern has been described in [13]. This pattern is also known as Threadpool pattern and Background Thread pattern.

Issue

By using a forking technique to create child processes upon a request arrival, performance is impeded: initialization of all attributes, like module instances, for each child process results in significant overhead. As system memory consumption is correlated to the amount of active connections, the total amount of connections is (approximately) limited by the RAM size divided by the memory needed for one connection. Therefore complexity of process initialization and its memory consumption need to be reduced.

Assumptions/Constraints

It is assumed that thread-pool processes do not conflict. Furthermore, all logic executed by the threads, e.g. modules, must be thread-safe[20] in order to prevent issues with shared data.

Solution

The Threadpool pattern is used in order to spawn Threadpool processes, which on their turn regulate a pre-defined amount of (worker) threads. This is performed before any request arrives in the server, so they can be instantly served. The amount of processes to be spawned and amount of threads to be initialized per process is configured in an external configuration file. Furthermore, HTTPD assesses the total number of idle threads in all processes, and forks or kills processes to keep this number within the minimum and maximum boundaries.

[19] http://httpd.apache.org/docs/2.0/mod/worker.html [Online accessed; 05.02.2012]
[20] http://en.wikipedia.org/wiki/Thread_safety [Online accessed; 05.02.2012]

The amount of Threadpool processes is controlled by the master server process in order to ensure that a request can always be served instantly. Each child process contains a listener thread, which listens to incoming requests, and a pre-defined amount of worker threads. Multiple processes and therefore multiple listener threads are mutually excluded by means of the accept mutex. This is performed in the same way as when the Preforking technique is used.

The listener thread only applies for the accept mutex if worker threads are available (in the idle queue) to process the request. When a listener thread receives a request, the request is placed in a job queue, in order to be processed by a worker thread. After completion of a request, the worker thread registers itself with the idle queue to inform the listener thread of being available for request processing.

The HTTPD has various configuration parameters that are used during server startup and process management. Data structures are maintained for process accounting by the Master Server. These structures are stored on a shared memory with the form of a table called scoreboard.

Using the latter, the master process can investigate the amount of available (idle) threads, in order to kill them and start new. That way the desired amount of idle threads is retained. Based also on scoreboard, processes are killed and replaced after their limited lifetime.

Rationale

It is inefficient to wait for incoming requests and spawn new processes, therefore processes are spawned before requests are received. However, using a process to answer an incoming request results in inefficient resource utilization, as processes have large memory footprints, compared to threads.

Implications

- Threadpooling enhances the overall performance of the server and increases the amount of requests that can be processed concurrently. This is achieved by reducing the overhead in RAM utilization, which is result of the limited memory footprint needed for threads.
- Security is enhanced because of the killing of processes due to their limited lifetime. This results in protection of compromised processes.
- However by sharing resources among threads within the same pool, a single thread that produces malicious results can have disastrous effect on all threads in its pool. Though this disadvantage on security and reliability is acceptable, since this pattern focuses on performance.
- Threadpooling leads to a more complex architecture. This could mean less portability, as well as more faults.

Related Patterns

- Leader/Followers 5.4.1
- Secure Pre-Forking 5.4.2

6 Evaluation

In this section the documented patterns, subsystems and the system as a whole are evaluated against the key drivers of the architecture (cf. Section 3.3). For the evaluation, the lightweight review process provided by PBAR [5] is used, along with Force Resolution Maps (FRM). The FRM method offers a simple representation of the outcome of the evaluation. The scale used in the FRM ranges from -2 to +2, as illustrated in Table 2.

Table 2: Definition of the FRM values

Value	Definition
-2	Severe negative impact
-1	Small negative impact
0	Non-affected or neutral, both negative and positive
+1	Small positive impact
+2	High positive impact

6.1 Patterns

In this section the identified patterns are evaluated with respect to their impact on the key drivers.

6.1.1 Pipes and Filters

The implementation of the Pipes and Filters pattern provides the possibility of variable composition of the filter chains. This means that filters, in this case modules, can be easily added, removed and rearranged in any order that is required for certain processes. Hence the extensibility is greatly improved (+2).
However, it is not really efficient that data packages must be transferred from filter to filter, even if only a part of them will be changed. This causes a slight decrease in efficiency (-1).
Pipes and Filters pattern does not affect security. Its implementation neither poses any additional security issues, nor enhances security.
Portability is also not affected as this pattern is realized in the core.

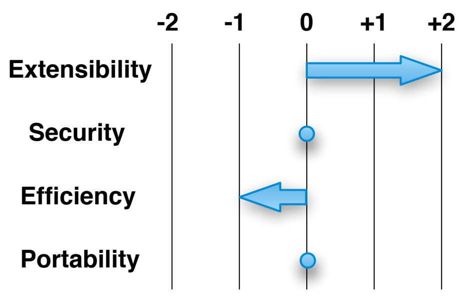

Figure 18: FRM for Pipes and Filters pattern

6.1.2 Shared Repository

The Shared Repository uses only the fast RAM memory (cf. Section 5.1.2). With the implementation of it, the data are not forwarded between the filters, but instead they access the data as needed. The filters

pass references of the data, therefore next filters are able to access them in memory. Hence, efficiency is greatly improved (+2). The impact on the key driver is depicted in Figure 19.
The exchange of references between filters provides an abstract way of accessing the data. This is the reason that security is enhanced. The data of the requests are protected by residing in the memory, instead of passing through filters during request processing. Memory usage could pose security issues. However as Shared Repository is implemented in a process and processes have strict memory to use, it is obliged to use only this memory.
Although, the Shared Repository pattern is used in combination with Pipes and Filters, it has no effect on the extensibility. Filters can be still easily added or removed. Portability is also not affected as this pattern is realized in the core and it uses the APR in order to abstract operating system depended calls.

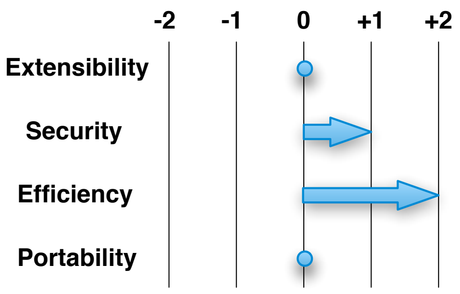

Figure 19: FRM for the Shared Repository pattern

6.1.3 Interceptor

The Interceptor pattern realizes the hook mechanism that Apache HTTPD provides. It increases extensibility (+1) of the server because processes can be easily added or extended. For example, performing any operation on each incoming requests is easily achievable by using the hooking mechanism.
Efficiency is slightly impeded (-1). The degree to which efficiency is affected depends on the number of handlers that have to be executed. If a lot of handlers have to be executed, this could reduce the efficiency of the system. Also the hook executor introduces an extra layer of indirection , which creates additional overhead due to an increased number of function calls.
Portability is not affected since the Interceptor pattern neither helps nor hinders the way in which modules must be treated with regard to the operating system. It still allows the modules to take advantage of the APR, which is how Apache server mainly deals with portability. Security is also not affected, because the modules are responsible for implementing security measures. The fact that modules cannot be trusted always is taken into account during the evaluation of the Plugin pattern, thus security remains unaffected.

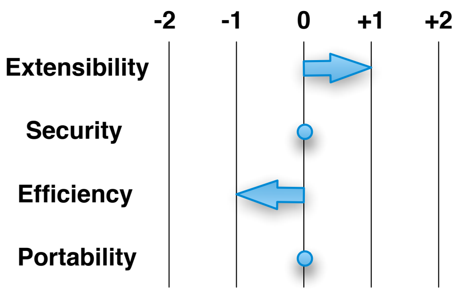

Figure 20: FRM for Interceptor pattern

6.1.4 Plugin

It greatly enhances extensibility (+2) by allowing the loading of modules during runtime, as mentioned on section 4.1.2. It also increases the portability of HTTPD, since modules can be ported to all platforms independently, without affecting the server or other modules. Not all modules are available for all operating systems though, that is why (+1) is assigned instead of (+2). For example, the mpm_winnt module is specific for windows only.
As a consequence it slightly impedes efficiency (-1) due to additional code indirections. Also the modules have to be loaded, which takes time. Security is negatively affected (-1) due to the use of third party modules. Third party modules do not always function as intended, so this may pose a security threat for the server.

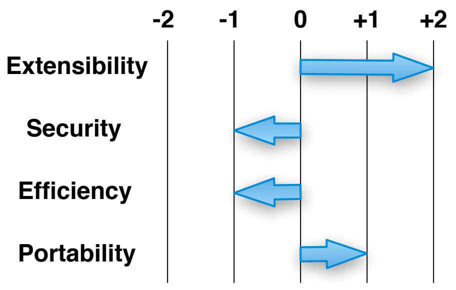

Figure 21: FRM for Plugin pattern

6.1.5 Indirection Layer

The Indirection Layer pattern is implemented to make it easier for developers to write solutions that support different operating systems like Windows or Linux. This decision means that portability is positively influenced (+2).
However, it is not mandatory to use this layer in order to access OS-depended functions. Nevertheless, it is allowed and even sometimes facilitated to bypass the Indirection Layer for performance critical operations. Hence, the efficiency is not negatively affected although there might be some additional indirections.
Security is enhanced (+1) because an extra layer with implemented security mechanism has been introduced. This layer can manage the memory access and control the way a process can or cannot access a memory space. This will prevent malicious or malfunctioning code from interfering with it. The Indirection Layer does not affect extensibility as it makes the whole package platform independent without affecting the application logic itself. This is illustrated in Figure 22.

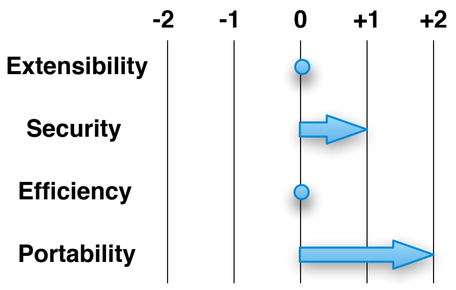

Figure 22: FRM for Indirection Layer pattern

6.1.6 Leader/Followers

The Leader/Followers pattern is implemented partially in the multiprocessing modules of Apache and partially in the core. The use of this pattern ensures that incoming requests are served instantly. This increases efficiency significantly (+2). Portability is negatively affected (-1) because the concrete implementation of this pattern bypasses the Apache APR in order to increase efficiency. The trade-off is the reduction of portability for the sake of efficiency.
This pattern does not affect extensibility since it is not related to it. Although this pattern introduces complexity in the code, complexity is not considered as a key driver of Apache. However the complexity of implementing multitasking architectures always poses security issues. Although security could be negatively affected it has a neutral impact on security, since the pattern itself and its concrete implementation do not introduce any potential risk besides a higher complexity.

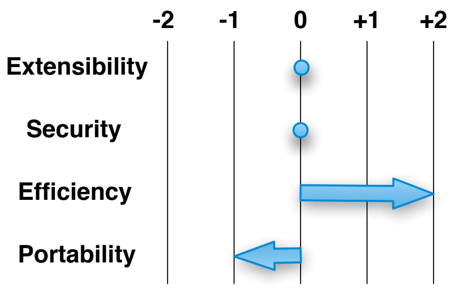

Figure 23: FRM for Leader/Followers pattern

6.1.7 Secure Pre-Forking

The Secure Pre-forking pattern is implemented in the mpm_prefork module of HTTPD. This pattern increases security (+1) by limiting the lifetime of processes. Due to this this limitation memory leaks are less likely to appear.
Forking is a technique to utilize multi-core capabilities of the hardware. By forking the processes, multiple requests can be processed by the hardware simultaneously, which will increase the efficiency drastically. On the other hand efficiency is also decreased, because resources are not efficiently utilized. Spawning new processes is a costly operation resource-wise. Thus, the killing and spawning of processes is inefficient, which decreases the impact on efficiency. All these result to (+1) impact for efficiency.
As with the Leader/Followers pattern, the complexity of the code is increased. It does not affect extensibility, since it is implemented in a module. Portability is also not affected because process management is implemented by the Leader/Followers pattern.

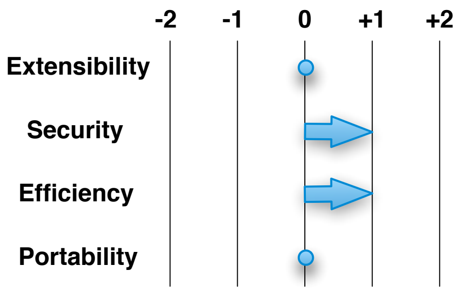

Figure 24: FRM for Secure Pre-forking pattern

6.1.8 Threadpool

The Threadpool pattern is implemented in the mpm_worker module of HTTPD. As mpm_worker is a (compile-time) module and can be left-out or replaced, it has no effect on extensibility or portability of Apache HTTPD.
Threads share virtually all resources like module instances, which allows a malicious thread to infect other threads in its pool. This results in decrease of security. Although reliability is not a key-driver it is reduced for the same reason: one crashed thread can cause a chain reaction on other threads in its pool. As the module typically starts multiple (distinct) threadpools, which cannot directly access one another's resources, a faulty thread will not be able to infect/crash the whole system.
Due to this mechanism, the total security impact could be rated negatively. However processes have limited lifetime, and the killing of them, in which the threads exist, enhances the security. This is the reason of a neutral impact on security.
This pattern has a very positive effect on efficiency, as it utilizes the multi-processing capabilities of the hardware. The memory footprint is reduced to a minimum, as threads merely own a separate execution stack. Overhead on memory is minimized, therefore it has no significant consequences on efficiency. Altogether this pattern has a very positive (+2) influence on efficiency.

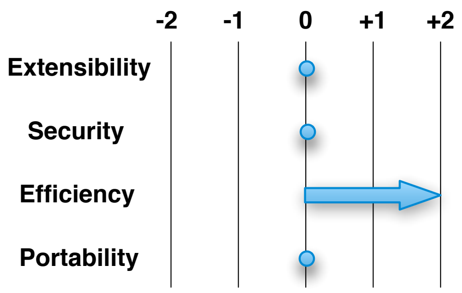

Figure 25: FRM for Thread Pool pattern

6.2 Subsystems

In this section the subsystems are evaluated with respect to their impact on the key drivers.

6.2.1 Core Subsystem

In this section, the core subsystem will be discussed. Part of this subsystem are the Pipes and Filters pattern which is used in combination with the Shared Repository pattern. As described in Section 6.1.1 it is not efficient to copy data packages from one filter's output to another filter's input. However, with the cooperation of a shared repository this is efficiently addressed. Combining the two patterns will result in a positive (+1) impact on efficiency.
Security was enhanced by the Shared Repository. As Pipes and Filters pattern does not affect this key-driver, this subsystem is evaluated positively (+1) for security. Extensibility was increased by Pipes and Filters, due to the possibility to add or remove a module (acting as filters), as well as rearranging the order of them in every process. Hence, the core subsystem has high positive impact on extensibility (+2). The last key driver is portability, but neither the Pipes and Filters nor the Shared Repository affect it.

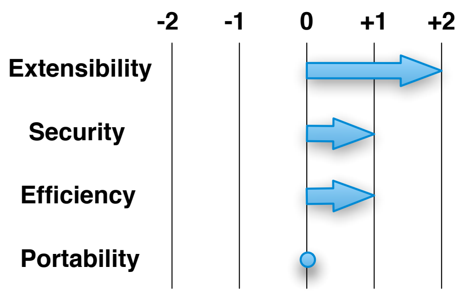

Figure 26: FRM for the core subsystem

6.2.2 Modules Subsystem

In this subsystem of HTTPD the Interceptor and Plugin patterns are used. Extensibility is greatly enhanced (+2) for this subsystem. The hooking mechanism along with the inclusion of modules during runtime, results in a facile way of adding extra functionality. The portability of the subsystem is also increased (+1) because modules can be ported between different platforms without affecting other modules or the system.
On the negative impacts, security is decreased (-1) because of the use of third party modules which might pose a security threat. Efficiency is also impeded (-1). The loading of modules along with the extra overhead that is created from the Plugin and the Interceptor patterns, results in a negative impact on efficiency.

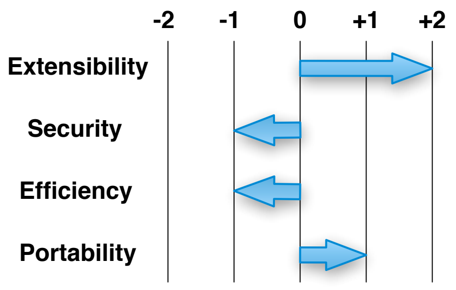

Figure 27: FRM for the modules subsystem

6.2.3 Apache Portable Runtime Subsystem

This section describes the pattern presented in the Apache Portable Runtime subsystem. This subsystem involves only the Indirection Layer pattern which is already discussed in Section 6.1.5. Hence, the FRM of the subsystem is exactly the same as the FRM of the pattern.

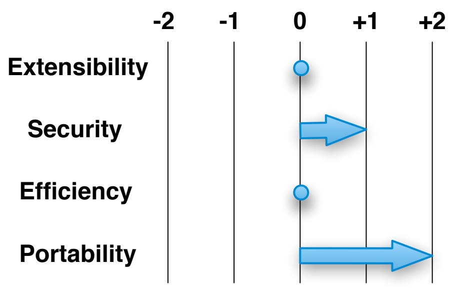

Figure 28: FRM for the APR subsystem

6.2.4 Multiprocessing Subsystem

For this subsystem two FRMs are present, one for the mpm_prefork and one for the mpm_worker configuration. The reason for this is that only one of the modules can be active at the same time. First the FRM for the mpm_prefork.

In the mpm_prefork module, the Secure Pre-forking and the Leader/Followers patterns are used. Security is positively affected (+1) because memory leaks, which could pose security threats, are taken care of by recycling of processes. Efficiency is also increased (+1). It is not (+2) because Secure Pre-Forking is not very efficient as far as memory resource utilization is concerned.
Portability is negatively affected (-1) because the Leader/Followers pattern bypasses the APR for more efficiency. Extensibility is not affected as neither of the patterns involved, are related to it.

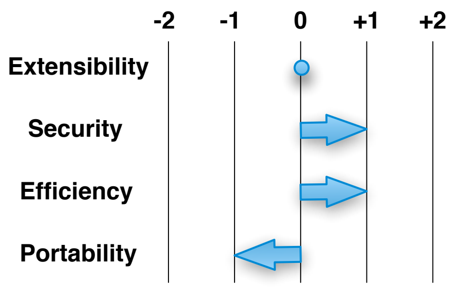

Figure 29: FRM for the multiprocessing subsystem using mpm_prefork

In the mpm_worker module, the Threadpool and the Leader/Followers patterns are implemented. This gives the worker version of this subsystem a boost in efficiency (+2). Efficiency is increased by sharing data and module instances between threads as described earlier (cf Section 4.1.4).
Portability is negatively affected (-1) for the same reason as the mpm_prefork module. Security is neither positively nor negatively affected because on the one hand, there is the sharing of data and module instances between threads. If an error occurs in a thread, this could also affect other threads. On the other hand, the killing of processes due to their limited lifetime increases security.
Again extensibility is not affected by this subsystem for the same reason as before.

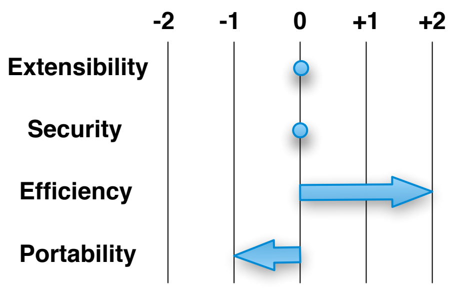

Figure 30: FRM for the multiprocessing subsystem using mpm_worker

6.3 Overall System

For the overall system two FRMs are provided, one for the secure pre-fork version and one for the worker version. Firstly the overall system configured with mpm_prefork is described.

Extensibility is greatly enhanced (+2) due to the core and the modules subsystems, respectively the Plugin pattern. They are both positively contributing in a different way to the extensibility of the system, as already mentioned in the preceding FRMs. Portability is positively affected (+1) mainly due to the APR which decouples operating system depended calls.
Security is also positively affected (+1) mainly due to the recycling of processes. As already mentioned, this drastically decreases the negative effect that a compromised process poses. However a trade-off between security and efficiency is presented due to the frequent re-spawning of processes. This trade-off results to a neutralized impact on efficiency for this version of the system. It is not (+1) as in Section 6.2.4, because the effect of this issue in the overall system is higher.

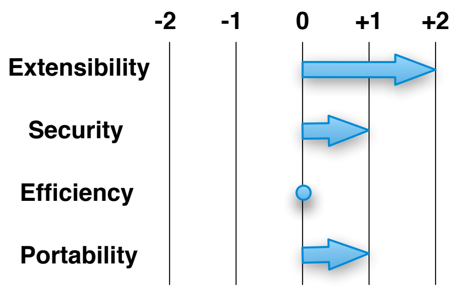

Figure 31: FRM for the overall system using mpm_prefork

The evaluation for the overall system using the mpm_worker, has a positive effect on extensibility for the same reasons as with the mpm_prefork version. Efficiency is also positively affected (+1) due to the resource utilization provided by the multiprocessing subsystem. Portability for this version, is also positively affected (+1) for the same reason as with the Secure Pre-forking version.
Security is neutral for this version of the system, mainly because of the mpm_worker module. The probability of a thread crashing and starting a chain reaction, is definitely a security issue. However, the system is protected due to the recycling of processes. Thus, a neutral impact is assigned to the overall system.

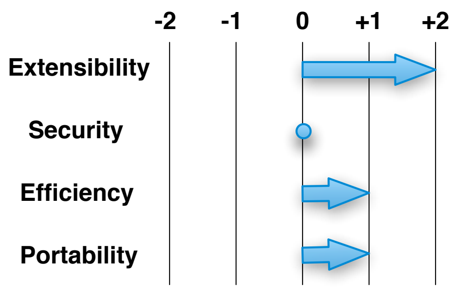

Figure 32: FRM for the overall system using mpm_worker

6.4 Quality Attribute Pattern Matrix

In order to sum up the evaluation, the results of the FRM evaluation is presented in a compact matrix. The values of the matrix are the same as the values for the FRM.

Table 3: Quality Attribute Pattern Matrix

	Extensibility	Security	Efficiency	Portability
Core	++	+	+	/
Pipes and Filter	++	/	−	/
Shared Repository	/	+	++	/
Modules	++	−	−	+
Interceptor	+	/	−	/
Plugin	++	−	−	+
Apache Portable Runtime	/	+	/	++
Indirection Layer	/	+	/	++
Multiprocessing (prefork)	/	+	+	−
Multiprocessing (worker)	/	/	++	−
Leader/Followers	/	/	++	−
Secure Pre-Forking	/	+	+	/
Threadpool	/	/	++	/
System (prefork)	++	+	/	+
System (worker)	++	/	+	+

7 Recommendations

The evaluation shows that the used patterns result in conflicting or negatively addressed quality attributes. However, the modularity of the Apache HTTPD allows the user to configure the server according to his needs. As illustrated in Table 3, the pre-fork based Apache is more secure but suffers in efficiency, while the worker based configuration trade-offs security in favor of efficiency.

The emphasis on extensibility and flexibility has been clearly identified during the evaluation, which is conform to the assumed priority of key drivers in Section 3.3. Indeed, the extensibility is so important that security has been sacrificed to some extent.
Besides the conflict between security and efficiency, as well as between security and extensibility, there is also a conflict between portability and efficiency. The concrete implementation of the Leader/Followers scheme bypasses the Indirection Layer in order to boost the request processing performance. This trade-offs the portability of Apache because not only the APR has to be ported but also the multiprocessing modules.

Due to the large amount and specialization of modules only the most important modules have been explored for patterns. This includes the modules responsible for multi-processing and the core.

In the following, some recommendations are given to increase efficiency, portability or security.

7.1 Event-driven

Apache HTTPD is one of the oldest webserver that is still used and maintained. At the beginning of HTTPD's development, it can be speculated that portability might not have been an important key driver. Making HTTPD portable would definitely not have been as high-valued as performance in the early days, when IIS still had an hefty market share. Currenlty, HTTPD is ported with similar performance to every platform.

Nowadays Virtual Private Servers (VPS) are often used for hosting web-sites. A VPS is a virtual machine, typically running on some (cloud) virtualization server. The amount of RAM on these machines is usually very limited, and can even be as low as 128MB for example. A VPS with this kind of memory size can run Apache, but will quickly exceed the memory limit because neither the worker nor the pre fork configuration are very efficient in terms of resource utilization.

Competitors of Apache HTTPD, like Nginx and Ligthttpd, have been well received as a proper alternative by the community. The popularity of these competing webservers is accreditable to their low memory usage under heavy load.

The difference between Apache HTTPD and Lighttpd-Nginx, is that the latter are asynchronous servers, and the former is a process oriented server. The main advantage of the asynchronous approach is that it is more efficient, as the asynchronous approach is event-driven and handles different stages of processing in multiple threads. In contrast to Apache HTTPD, they require a thread to finish processing from beginning to end before it can accept a new connection. This is inefficient because the time Apache HTTPD takes to wait for PHP is time that is wasted, since this time can be used to accept a new connection.

In light of these arguments a recommendation to provide a solution to the VPS efficiency issues is obvious. An event-based multiprocessing module is recommend to Apache HTTPD. This would result in less memory consumption, as support threads - performing content-handler tasks - could serve multiple requests. It does however mean that modules in Apache need to support threading, which is not the case for most modules.

In regard to the key drivers, the Extensibility is reduced, as modules need to have threading capabilities. Efficiency would be increased tremendously. Portability and Security are not likely to change.

7.2 Deprecation of Modules

At this time, the Apache can be configured using proxy architecture. This means that Apache HTTPD can relay content-handler functions to their respective servers, instead to hooked filters. For example, PHP has the capability of running it's own multi-threaded server. It turns out that this approach is far faster and efficient than handling PHP through Apache HTTPD's PHP module.

It is faster because of the previous mentioned problems Apache HTTPD has, with threading their modules for use with an event-base multiprocessing. But it basically means that a chunk of functionality and it's risks are off-loaded to other software.

With the increased use of content-handler modules like php or python, Apache found themselves trapped by their own modules. This is because at this point, mod-php and mod-python are still heavily used as defaults. It is recommended that Apache enforces to deprecate these modules, or at least that their use for new installations is advised against.

If followed through, there is little effect of this change to the key drivers. Security is higher, as there will be less functionality in the modules, which reduces risks. Extensibility actually increases, as developers do not have to worry about older modules anymore and can focus on new functionality instead.

When content handlers would function as an external system, multiprocessing based on a threadpool architecture, would be more reliable. As mentioned in Section 6.1.8, this architecture has the drawback of a chain reaction in case of a thread crash. If content handlers are used as external servers, then this drawback is eliminated, resulting in enhanced reliability.

7.3 Virtual Machine

An recommendation to increase portability and security would be to upgrade the Indirection Layer by a Virtual Machine Pattern. Hence, the Apache would be completely decoupled from the host operating system.

Since this would have a positive impact on the portability of the system, the efficiency will decrease due to the additional indirection. However, a Virtual Machine offers the opportuninty of optimizations during runtime, which are not available in a static environment. Another aspect is, that the dynamic module loading does not depend on the platform anymore and thus all platforms can benefit from dynamic loading.

Besides the impact on portability, security is also positively affected because the Apache does directly access the Operating System and it is easy to implement additional security measures.

The Virtual Machine facilitates the Reflection Pattern, which would increase the extensibility of the platform (cf. [9]).

Nevertheless, the recommendation is difficult to implement and would require a major adaption of the Apache HTTPD. A more practical approach would be to optimize the Apache server to work with a particular existing Virtual Machine (e.g. Virtualbox or ZEN).

8 Conclusion

This project's goal was to review the architecture of the Apache HTTP Server. To accomplish this, software patterns have been identified and they have been evaluated according on their impact on the Software Quality Attributes. For reviewing the architecture, the Pattern-Based Architecture Reviews has been used. The IDAPO process has been used to identify the patterns in the architecture.

8.1 Pattern-Based Architecture Reviews

A complete evaluation of an architecture takes a lot of effort, especially if documentation is insufficient. In order to reduce the effort, the PBAR method has been developed. PBAR concentrates on issues or problems in relation to the system's important quality attributes. It uses discovered patterns of the architecture to properly document issues. A closer look to the exact process can be seen in [5].
The following enumeration explains how the steps of the PBAR process relate to our project.

1. **Resources and Planning:** This project has six reviewers allocated.
2. **Preperation:** Before starting, the entire team studied and gathered materials relevant to identifying patterns and derive quality attributes. The material consisted of the documentation of Apache, a modeling project for an older version of Apache, as well as other various internet resources. Unfortunately no resource stated the major requirements or most important quality attributes.
3. **Review Meeting:** PBAR states a face-to-face review needs to be done. Needless to say in this school-project we could not plan, or take a developers time away. Instead we conducted review meetings by meeting regularly as a team. The different parts of the review meeting were done repeated, for example the architecture and the stakeholder concerns and quality attributes were found to be stable in week three. A big review meeting has been conducted in the end, in which the impact of the patterns has been discussed ultimately and the issues and recommendations have been finalized.
    1. **Review QA Reqs:** In the first meetings, tasks were assigned to designate stakeholders and their concerns. This is documented in Section 3 and is the group's viewpoint about important requirements of the system and the most important quality attributes.
    2. **Review Architecture:** Based on found resources, the architecture was studied and a summary has been written in Section 4. This chapter serves as an introduction to the explanation of the patterns identified.
    3. **Identify Patterns:** The patterns were identified using the IDAPO method, which is further explained in the next section. The patterns are documented extensively in Section 5.1.
    4. **Study Pattern-QA Interactions:** Each pattern affects quality attributes in some way. This has been evaluated in the form of discussion many times and the results are documented in Section 6.
    5. **Identify Issues:** The prime directive with this project was to uncover critical patterns and to ultimately in a fast way understand Apache HTTPD's architecture. However, issues relating to QA interactions are evaluated, and based on this we have come up with a number of possible recommendations. These recommendations can be found in Section 7.
4. **Followup:** The entire team did frequent further analysis on the documentation and continued discussions via email, until the next meeting was scheduled.

The PBAR process is useful for small projects, where developers can meet with the reviewers to provide inside-information. For us, the most important link to PBAR and our project was the relationship between architectural patterns and quality attributes. They provided an optimal basis on which to further understand the Apache HTTP Server Project.

8.2 IDAPO

The IDAPO process is used to identify patterns in a software system. Even if developers may not be aware of it, patterns may still be present. Knowing what patterns are used in your software system is very useful to quantify your software project on a higher-level, and to express the system more easily in quality attributes.

The IDAPO process is like a roadmap that expresses different roads to the same end-goal, the identification of a software pattern. There are numerous steps involved in IDAPO and they are best explained in [4]. Instead of going over the IDAPO process step-by-step we will try to explain how our own process fits into IDAPO.

The team immersed themselves in the world of Apache HTTPD by trying to answer the question: What is Apache HTTPD and for what is it used? This way we familiarized ourselves with the project type and domain before starting identification of patterns.

It turned out that Apache HTTPD is a very large system, and although documented, in some critical inner workings it lacks of detailed documentation. Even with the help of the modeling project [2] that provided documentation for an older version of Apache, it was hard to distinguish which parts of the system were of significant importance.
Fortunately, the previous assignment provided sufficient experience about software patterns in general to assess wether something can be considered as a pattern or not. With this knowledge and the growing debates and discussions in our team, several candidate patterns have been successfully identified. Afterwards, these candidate patterns were evaluated and re-evaluated until enough evidence has been found to verify the presence of a certain pattern.

Source code has been examined to validate the presence of patterns. The community was interviewed briefly to ensure that key drivers were well established, and also to verify how the Apache architecture works in regards to its modular approach. A complete log can be found in Appendix A1.

In summary, the IDAPO process provides a very natural way in identifying patterns and was a big help for our project.

8.3 Final Words

The reason for choosing Apache HTTPD was to choose a popular, interesting and non-trivial system. We were especially impressed by Apache's modular approach to programming, and how this particular project aims to satisfy all users by providing multiple ways to achieve the same functionality. For example, Apache offers multiple ways of implementing multi-threading, one for the user that is concerned about compatibility with older software, and one that is more concerned for raw performance. As the modular approach gives users a choice, we decided to also give our readers a choice by giving multiple final evaluations with the use of different modules.

The experience we gained with the first assignment of Software Patterns payed off in this second assignment. By using that experience we were able to identify patterns easier, as we had more knowledge of their appearance. This reflected in the way we worked as a team, as misunderstandings were noticeably less frequent than in the first assignment.

The most important result of this project is that each of us now has a lot more knowledge about software patterns and Apache HTTPD's architecture. The cool thing is that we accomplished this without having to look at thousands of lines of code. The conclusion is that the use of software patterns and quality attributes contribute to a fast review of a software system, and a good understanding of the system's inner workings. The use of the IDAPO and PBAR process is well understood by each team-member and is likely to be used in future reviews of software systems. Along with this experience we have also written down a number of recommendations that could help Apache. We would like to thank Apache for their documentation and welcome community, our coach Matthias Galster for his valuable feedback during the whole process and Paris Avgeriou for the enlightening lectures about Software Patterns.

References

[1] "January 2012 web server survey," January 2012. [Online]. Available: http://news.netcraft.com/archives/2012/01/03/january-2012-web-server-survey.html

[2] B. Gröne, A. Knöpfel, R. Kugel, and O. Schmidt, "The apache modelling project," January 2008. [Online]. Available: www.fmc-modeling.org/download/projects/apache/the_apache_modelling_project.pdf

[3] "Ieee std 9126-1 information technology - software produce quality, part 1: Quality model," IEEE, 2000.

[4] K.-J. Stol and P. Avgeriou, "Idapo: A process for identifying architectural patterns in open source software," November 2011.

[5] N. Harrison and P. Avgeriou, "Using pattern-based architecture reviews to detect quality attribute issues – an exploratory study," 2011.

[6] "Apache project guidelines." [Online]. Available: http://httpd.Apache.org/dev/guidelines.html

[7] N. B. Harrison, P. Avgeriou, and U. Zdun, "Using patterns to capture architectural decisions," *IEEE Software*, vol. 24, no. 4, pp. 38–45, 2007.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, 1st ed. Wiley, 1996.

[9] P. Avgeriou and U. Zdun, "Architectural patterns revisited - a pattern language," 2005.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[11] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2.* Wiley, 2000.

[12] M. Hafiz, "Secure pre-forking - a pattern for performance and security," 2005.

[13] "Worker thread pattern java." [Online]. Available: http://www.eecho.info/Echo/design-pattern/worker-thread-pattern-java

[14] M. M. Corporation, "Load-balanced cluster," 2003. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff648960.aspx

[15] T. Erl, *SOA Design Patterns*, 1st ed. Prentice Hall/PearsonPTR, 2009.

[16] M. Fowler, *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

[17] I. Sommerville, *Software Engineering: (Update) (8th Edition) (International Computer Science Series)*, 8th ed. Addison Wesley, 2006.

[18] "Ieee std 830-1998 recommended practice for software requirements specifications," IEEE, 1998.

[19] M. Corporation, "Failover cluster," Website, 2003. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff650328.aspx

Appendix

A1 IRC logs with community

On #httpd at Freenode. Werner Buck is using the nickname "shadylog"

```
[20:32] <shadylog> Hi, i am part of a group that is currently using apache as a subject
                   to research it’s use of software patterns. We are encouraged to contact
                   the community to find out more on the subject
[20:33] <shadylog> Among other things, key drivers are used to quantify the top-priority
                   quality attributes of a software project. For apache we have found the
                   following keydrivers, "Extensibility", "Security", "Efficiency" and
                   "Portability". Do you as experts feel this is an accurate  representation
                   for apache’s key drivers?
[20:33] <shadylog> Also what kind of patterns do you think apache has?
[20:35] <thumbs>   shadylog: what apache project are you referring to?
[20:36] <shadylog> The apache HTTP server
[20:36] <fajita>   idesigned to be a powerful ... isn’t everything? (No discussion about
                   company’s failing to achieve though)
[20:36] <thumbs>   shadylog: then those keywords are a fine representation of the project, yes.
[20:36] <thumbs>   shadylog: secondly define "patterns"
[20:37] <shadylog> thanks~
[20:37] == njbair [~njbair@user-12l369a.cable.mindspring.com] has quit [Quit: Leaving]
[20:38] <shadylog> Well it’s hard to solidly define a pattern, but it can be anything from
                   the use of camelcase to model view controller
[20:39] <shadylog> they can be low-level, but we rather define high-level patterns as
                   the apache project is so large
[20:39] <shadylog> And we use http://www.google.nl/search?q=Architectural+Patterns+Revisited
[20:39] <shadylog> That article for general patterns
[20:40] <shadylog> think of "Broker","Pipes & Filters","Shared Repository"
[20:40] <shadylog> now we have already identified a bunch of patterns
[20:40] <thumbs>   shadylog: for that question, best ask #httpd-dev
[20:41] <shadylog> ok thanks.
[20:41] <shadylog> well thanks for your time :)
```

On #httpd-dev at Freenode.

```
[Chewing on the internals; http://httpd.apache.org/dev/]
[20:41] == shadylog [53755481@gateway/web/freenode/ip.83.117.84.129] has joined #httpd-dev
[20:41] <shadylog> Hi, i am part of a group that is currently using apache
                   as a subject to research it’s use of software patterns.
                   We are encouraged to contact the community to find out
                   more on the subject.
[20:42] <shadylog> So i was redirected from #httpd to here, because what i
                   want to ask are some architecture-related questions about
                   software patterns Apache HTTP server uses
[20:42] <jMCg>     shadylog: Apache’s software in general, or the HTTP Server in particular?
[20:42] <shadylog> Apache HTTP Server
[20:42] <jMCg>     ACK
[20:43] <shadylog> Acknowledged? :)
[20:43] <shadylog> Now we already are in our final phase of our project so
                   we have identified several patterns. But one thing still keeps us
                   somewhat in the dark
[20:44] <shadylog> we basicly define apache as Core+Modules
[20:44] <shadylog> and that everything is basicly from version 1.3 moved
                   to Modules to keep apache more flexible
[20:44] <shadylog> but what does the core do?
[20:44] <jMCg>     A bunch of things.
```

```
[20:45] <shadylog> like startup/shutdown/reload?
[20:45] <shadylog> because the MPM modules we gather do a lot of the process related stuff no?
[20:45] <jMCg>     Yes, it drives the loading of the modules, and by 2.3 that includes the MPMs
[20:46] <jMCg>     The line of what we put in core and what we put into modules
                   has always been a little hazy.
[20:46] <shadylog> Right
[20:46] <shadylog> Well we were trying to summarize the architecture for apache,
                   but couldn't find any concrete tasks for the core
[20:47] <shadylog> but is it fair to say that the hook management is done in the core?
[20:47] <jMCg>     Most of the time when new developers came along and decided
                   that this stuff is useless in core, or is better off in a module,
                   or vice-versa, they'd go ahead and just do it.
[20:48] <jMCg>     shadylog: yes, and from what I gather from discussions
                   between pquerna and sfritsch that's a problem for moving
                   ahead with mpm event.
[20:48] <shadylog> Hmm
[20:48] <shadylog> Yes MPM event looks very promising
[20:48] <jMCg>     I think they can explain it best. Or just look at Paul's
                   recent changes to trunk.
[20:48] <shadylog> thanks i will!
[20:49] <jMCg>     mpm event is an awesome step, and the architectural ideas
                   comes from Tomcat's processing model.
[20:50] <jMCg>     shadylog: you might want to look at Tomcat's docs.
                   They've got some sort of processing map.. oh, and then come back
                   and draw one for httpd :P
[20:50] <shadylog> haha
[20:50] <shadylog> I didn't realize tomcat was actually used in such an extent
[20:51] <jMCg>     I was trying to produce one for Traffic Server and using
                   Valgrind, but unfortunately it drove me insane. (I'm currently wearing
                   a straight jacket, typing with my tongue)
[20:52] <shadylog> personally i rather not touch valgrind unless i really really
                   have to, watching the output destroys braincells just as much as black
                   label whiskey
[20:52] <shadylog> ok, on a different note
[20:52] <shadylog> with MPM event, the memory usage would be lower correct?
[20:53] <jMCg>     shadylog: given that it's threaded, yes.
[20:53] <shadylog> as apache would take a more "on-demand" approach rather
                   than forking every time a new request comes in?
[20:53] <jMCg>     That always depends on how you configure it.
[20:53] <shadylog> hmm
[20:54] <jMCg>     And httpd doesn't fork for every new request either, but again,
                   that depends on how tight your configuration is.
[20:54] <jMCg>     see
[20:54] <jMCg>     20:59:15 <+fajita> http://httpd.apache.org/docs/current/misc/perf-tuning.html
                   or see also performance testing, or
                   http://wiki.apache.org/httpd/PerformanceScalingUp
[20:54] <jMCg>     The wiki, which I still haven't migrated to XML :-/
[20:54] <shadylog> nice :)
[20:54] <shadylog> but that is i assume over static content?
[20:55] <jMCg>     shadylog: this is a newer document, it doesn't only consider static content.
[20:55] <shadylog> how does it handle php loading?
[20:55] <shadylog> ah
[20:55] <jMCg>     One way or the other, with event you wouldn't want PHP to be a module.
[20:55] <shadylog> yes how would php be handled in mpm-event?
[20:56] <jMCg>     You'd want to proxy to it, or you'd use mod_fcgid, or similar. Whoa.
[20:56] <jMCg>     mmmhmmmm...........
[20:56] <jMCg>     I'm a genius.
```

```
[20:56] <shadylog> :P
[20:56] <jMCg>     PHP 5.4 comes with a built-in server. You could spin a couple of those
                   up and proxy to them. But, that's probably insane.
[20:57] <shadylog> yeah php has it's own fcgi server does it not?
[20:57] <jMCg>     They'd be single process/single thread, I suppose... Dunno.
[20:57] <shadylog> they would i guess
[20:57] <jMCg>     shadylog: with mod_prox_fcgi you can also use fcgistarter and
                   stuff like that, so there's tons of methods these days to run PHP sensibly.
[20:58] <jMCg>     mod_php is a monstrosity which you don't want to poison your server with.
[20:58] <jMCg>     I do it none the less, but only in backends.
[20:58] <shadylog> right, using php as a module on a memory-tight server is not that smart
[20:58] <shadylog> yeah it only serves to ease setup of LAMP servers
[20:59] <shadylog> although i have often used mod-php :)
[20:59] <shadylog> So what is your take on what direction apache will take in
[20:59] <shadylog> is it in your opinion taking a lot of heat from lighttpd or nginx?
[21:00] <shadylog> is it fair to say they hold the cards when it comes to
                   raw performance/cost during mpm-event is still unstable?
[21:00] <shadylog> im just reaching for a conclusion in our document :)
[21:00] <shadylog> "What will apache do next" kinda thing :)
[21:01] <shadylog> (this whole conversation will be appendixed) :P
[21:01] <jMCg>     I might be the wrong kind of people for this :-/
[21:01] <jMCg>     sfritsch? ping
[21:02] <sfritsch> shadylog: http://www.fmc-modeling.org/projects/apache
                   may be useful, even if it is for 2.0.
[21:03] <sfritsch> About the core: It calls the modules through hooks and provides
                   infrastructure for the modules (managing configuration, including which
                   parts of the config are applicaple to a request, logging, reading/writing
                   from/to the network, regexs, ...).
[21:04] <shadylog> sfritsch: we have already studied that entire document :)
[21:04] <sfritsch> About the future: I think trying to work on an async
                   (i.e. event based) solution for ssl is the next big project.
[21:04] <shadylog> Ah, i would have imagined configuration being something a module would do?
[21:05] <shadylog> nice!
[21:05] <sfritsch> The modules define configuration directives, but the core determines
                   which "Location" and "Directory" blocks should be used, merges them
                   and provides the result to the modules.
[21:06] <sfritsch> The content handlers will stay non-event driven a bit longer
                   than the connection/network processing. The current non-event mode
                   is way easier for module writers.
[21:07] <shadylog> ah
[21:07] <sfritsch> I think there was a mailing list thread on that, let me see...
[21:08] <shadylog> for example php-mod as a content handler would
                   stay non-event driven longer? Why not reccomend the jump
                   to proxying php fcgi servers altogether ?
[21:10] <shadylog> well i guess that's not fair, the php module is
                   by far the easiest to use and configure for most applications.
                   And if you really want to use apache you know you have
                   to get someone to properly configure it for you
[21:10] <sfritsch> Yes, for complex script interpreters, moving to a separate process
                   is better. But for example mod_autoindex is also a content handler,
                   and it would only complicate things to make it event driven.
[21:11] <shadylog> I see your point
[21:11] <jMCg>     shadylog: mod_wsgi
[21:12] <sfritsch> We hope that using php fcgi will get easier now that fcgi support
                   is included in httpd by default. mod_php is really bad in that it keeps a
                   huge php interpreter around for every request, even if the request
                   is only for some image file.
```

```
[21:12] <shadylog> right, i gathered this is what makes apache eat memory i guess
[21:12] <sfritsch> And it has big problems with threading because
                   many php plugins are not thread-safe.
[21:12] <jMCg>     I think I was trying to make that point somewhere.. here:
[21:12] <jMCg>     http://blag.esotericsystems.at/2010/02/introducing-more-caching/
[21:13] <shadylog> mod_wsgi, i only used it so far for python ?
[21:13] <jMCg>     "You might ask: Why" on that page.
[21:14] <shadylog> Why :)
[21:14] <sfritsch> jMCg: I thought mod_wsgi is similar to fcgi in that it creates a
                   socket connection to a separate python process. Am I wrong?
[21:15] <jMCg>     sfritsch: I must admit, I'm only a user.
[21:15] <shadylog> your right but i thought the difference was that fastcgi itself starts
                   a process. And wsgi starts the process for you?
[21:16] <shadylog> i'm not sure myself
[21:18] <shadylog> Anyway our group will take a look at those links you have provided :)
                   This was really helpfull and on behalf of our team we thank/complement
                   you guys for being so accessible and open!
[21:19] <sfritsch> Sorry, can't find the discussion about event, ssl, and async.
                   Maybe it was not on the list but during apachecon.
[21:20] <sfritsch> It depends on the implementation who starts the fcgi server process.
                   For example, mod_proxy_fcgi doesn't do it itself but expects some other
                   process manager to do the work (like the one built into php).
[21:20] <shadylog> That's ok :) we will look into the mailing lists for it
[21:22] <shadylog> nice, i haven't thought of using php's built in one
                   actually will take a look :)
```

A2 Process-Report Mapping Matrix

Table 4 illustrates the mapping of process steps to the related sections in the report.

PBAR	Report	IDAPO
-	Section 1	-
-	Section 2	1. Identify the type and domain of the product
1. Requirements and quality attributes	Section 3	
2. Architecture	Section 4	2. Identify used technologies
		3. Study used technologies
		4. Identify candidate patterns
		5. Read patterns literature
3. Architectural Patterns	Section 5	6. Study documentation
		7. Study source code
		8. Study components & connectors
		9. Identify patterns and variants
		10. Validate identified patterns
		11. Get feedback from community
		12. Register pattern usage
4. Impact on quality attributes	Section 6	-
5. Identify issues	Section 7	-
-	Section 8	-

Table 4: Mapping of Process Steps to Report Sections.

A3 Stakeholder-Concern Matrix

Table 5: Overview of stakeholders concerns with respect to the quality model [3]

	System Administrators (SH-01, High)	Apache HTTP Server PMC (SH-02, High)	Website Visitors (SH-03, High)	Web-browser developers (SH-04, High)	Website Developers (SH-05, Medium)	Apache HTTP Server Committers (SH-06, Medium)	ASF Board of Directors (SH-07, Low)	ASF Security Team (SH-08, Low)	Apache Developers (Contributors) (SH-09, Low)	Hackers (NSH-01, High)
Functionality										
Suitability										
Accuracy										
Interoperability										
Security	X		X	X	X	X		X	X	X
Reliability	X		X							
Maturity										
Fault Tolerance										
Recoverability										
Usability										
Understandability										
Learnability										
Operability										
Attractiveness										
Efficiency										
Time Behavior			X		X					
Resource Utilization	X									
Maintainability										
Analyzability								X		
Changeability		X				X	X		X	
Stability	X	X		X	X					
Testability										
Portability					X		X			
Adaptability	X									
Installability	X									
Co-existence	X									
Replaceability										

A4 Considered Patterns

A list of candidate patterns that have been truly considered but could not be approved.

A4.1 Microkernel

At first the Microkernel pattern has been considered to be present in the Architecture of the Apache HTTPD. The small core along with the modular approach looked similar to a Microkernel architecture.

However after further investigation, the external server APIs could not be identified and thus the forwarding of request to internal servers seems not to be a responsibility of the core.

After an in-depth investigation of the source code and the responsibilities of the core the presence of the Microkernel has been neglected.

A4.2 Client-Server

The Apache has been considered a Client-Server since the Apache is a Server and the Browsers are the clients. However, the Pattern is not explicitly part of the architecture of the Apache server and thus has not been further investigated beyond its proposal.

A4.3 Proxy

The Apache HTTPD can be seen as a Proxy. However, this has no influence on the architecture of the Apache since it is only one particular usage and configuration scenario. This pattern has not been further investigated after its proposal.

A5 Time Tracking

Week 1 16.12 - 22.12

Person	Task	Hours
cm	Setup document	1
	Prepare Meeting	1
	Update Outline, Share Todos, Cover Image	1.5
	Review Meeting	0.5
	Group Meeting	3
		7
si	Group Meeting	3
	Review Meeting	0.5
	Pattern and Architectural decision template	0.5
		4
jh	Intro Apache	0.5
	Review Meeting	0.5
	Group Meeting	3
		4
wb	Group Meeting	3
	Review meeting	0.5
	Apache googling	0.5
		4
fk	Group meeting	3
	Review meeting	0.5
		3.5
ah	Review Meeting	0.5
	Group Meeting	3
		3.5
	Group total	**26**

Week 2 23.12 - 05.01

Person	Task	Hours
cm	Writing Introduction	1.5
	Preparing Meeting	0.5
		2
si	Read Apache Documentations	1
	Key Drivers	1.5
		2.5
jh	Reading Documentation	2
	System Context	2.5
	Pattern Suggestions	1.5
		6
wb	Research Development for Apache, architecture	3
	Search Stakeholders	1
	Formulate stakeholders and concerns	4.5
	Read intro and sys context	0.5
		9
fk	Research on Apache server	2.5
		2.5
ah	Research on Apache httpd + candidate patterns	3
		3
	Group total	**25**

Week 3 06.01 - 12.01

Person	Task	Hours
cm	Group meeting	6
	Team building	8
	Research Microkernel, Preprocessor	2
	Meeting Preparation	1.5
		17.5
si	Read Apache Documentations	8
	Group meetings	6
	Document shared repository pattern	2
	Key drivers improvement	1
	Team building	8
	Pipes & Filters	1
		26
jh	Team Meeting 16.00 - 19.00	3
	Reading doc and minor adjustments	1
	Team meeting + working on stakeholders	3
	Group meeting	3
		10
wb	Group meeting	3
	Team building	8
	Group meeting	3
	Touched up Stakeholders and concerns	0.5
	Added interceptor traceability	0.2
		14.7
fk	Write about DSO and plugin pattern	2
	Read document and do minor corrections	1
	Group meeting	3
	Team building	8
	Group meeting	3
		17
ah	Group meeting	3
	Team building	8
	Elaborate patterns	2.5
	Group meeting	3
		16.5
	Group total	**101.7**

Week 4 13.01 - 19.01

Person	Task	Hours
cm	Review Meeting	1
	Presentation preparation	5
	Pattern documentation	8
	Group communications	1
	Review and correction of the complete document	9
		22
si	Shared Repository	1
	Secure Pre-forking	1.5
	Read Apache Documentations	3.5
	Leader/Follower	1
	Patterns Documentation Reading	2
	Coach meeting	1
	Review and correction	2
	Presentation preparation	9
		21
jh	Coach Meeting	2
	System Overview	1.5
	Stakeholders	2
	Review Doc	1.5
	Team Meeting	1
	Read Doc Group 1	1
	Project Management	0.5
		9.5
wb	Coach meeting	1
	Architecture Section	6
		7
fk	Work on keydrivers and stakeholders	2
	Coach meeting	1
	Review stakeholders and keydrivers	1
	Go through the document for minor fixes	1.5
	Spellcheck and concistency checks	2
	Read other groups document	3.5
		10.5
ah	Architecture: MPM	1.5
	Architecture	4.5
	Fix timetracking	0.5
	Reviewing & Correspondance	1
	Coach meeting	1
	Review document team 1	1
		9.5
	Group total	**79.5**

Week 5 20.01 - 26.01

Person	**Task**	**Hours**
cm	Improved patterns	2
	Split up architecture and pattern chapter	4
		6
si	Mapping IDAPO, PBAR	1
		1
jh	Evaluation Chapter	5.5
	System Overview	0.5
	Stakeholders	0.5
		6.5
wb	Reccomendation section, about half done	2
		2
fk	Work on evaluation chapter	5
		5
ah	Elaborate review + fixes to architecture	7.5
		7.5
	Group total	**28**

Week 6 27.01 - 02.02 **EXAMS**

Person	**Task**	**Hours**
cm	Prepare meeting	1
	Group communications	1
		2
si		0
		0
jh		
		0
wb		
		0
fk		0
		0
ah	Finish review	2
	Correspondence	0.5
		2.5
	Group total	**4.5**

Week 7 03.02 - 12.02

Person	Task	Hours
cm	Review of 6,7,8	4
	Recommendation	3
	Review of 1,2,3	3
	Check quality attributes against concerns	4.5
	Overall improvement	8
	Redesign of market share	3
		25.5
si	Diagrams FRM, Threadpool Pattern	3
	Reading Documentation	1.5
	Zoom diagrams - text	1
	Group Meeting	4
	Review Sec 5, 6	7.5
	Apache docs, code	1.5
		18.5
jh	Group meeting	4
	Evaluation	3
	Review Chapter 4 and 5 + Communication	3
	Review Chapter 5	1
	Review whole document	2.5
	Minutes	0.5
		14
wb	Community research	1
	Community research, IRC conversations with developers	2.5
	Community section finished in section 2	1
	Architecture Section - core defined and outlined 2	2
	Architecture Section - moved stuff around, completed section 4 2	1
	Recomendations	6
	Conclusion	4
		17.5
fk	Work on evaluation	8.25
	Group meeting	4
	Review chapter 4 and 5	3
	Review whole document	1.25
		16.5
ah	Add Worker Thread/Threadpool pattern	2
	Fix-up chapter 7	1
	Re-read & Fix-up chapter 6	2.5
	Correspondence	0.5
	Finish Threadpool pattern & FRM	1.5
	Review 4+5+6 again + Correspondence	7
	Review 7+8 again + Correspondence	3.5
		18
	Group total	**110**