Architecture recovery of Apache 1.3 — A case study

Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel

Hasso Plattner Institute for Software Systems Engineering
P.O.Box 900460, D–14440 Potsdam, Germany
E-mail: {groene, knoepfel, kugel}@hpi.uni-potsdam.de

Abstract

This document presents experiences from a course in which the authors taught students a way to understand and model software systems and share their knowledge about them. The real–life system examined in the course was the World Wide Web (WWW) and the Apache HTTP Server. The conceptual architecture of the system was modeled using the Fundamental Modeling Concepts (FMC) which turned out to be well suited for sharing knowledge about both concepts and details of the system. Excerpts of the model are presented in this document.

Keywords: Architecture recovery, Conceptual Architecture View, System Modeling, Fundamental Modeling Concepts

1. Introduction

Understanding existing software is an everyday task in software engineering. You often need to evaluate software products, e.g. if you join in or take over their development or if you just want to use them in your own project. If the complexity of a software product reaches a certain level, there is a need for division of labor requiring communication and for a systematic approach.

The curriculum for software systems engineering at the Hasso–Plattner–Institute (HPI) provides a practical seminar in the 4th semester, where students examine a real–life software product closely, acquire knowledge about it and present their results to the group. In 2001, the students examined the Apache 1.3 HTTP server.

Although everyone was familiar with the World Wide Web, getting a detailed conceptual architecture model of the system (including subjects like HTTP, DNS, Virtual Hosts and so on) took half of the semester. After that, the students had to examine the implementation of Apache. The conceptual architecture model of Apache developed in the course turned out to be very important for explaining both concepts and implementation design decisions. It was modeled using Fundamental Modeling Concepts (FMC, see section 2.2).

After the seminar, the conceptual architecture model of Apache was used for several presentations in industry. The material of the seminar has also been prepared for display on a web site and can be found at [4].

Section 2 of this document describes the structure of the seminar. An excerpt of the conceptual architecture of Apache is presented in section 3. In the conclusion, the authors present their experience with the seminar and with the use of FMC.

2. The Seminar

The idea behind the seminar was to teach 60 students a method of mastering the complexity of a software product. The students needed to understand the Apache 1.3 HTTP server and its implementation. We have chosen Apache because it is a real–life software product which is used all over the world, is actively developed, provides open sources and shows a certain level of complexity. The source code of Apache 1.3.17 consists of about 100.000 lines of C code, making it a rather small productive software product.

The authors assigned 32 topics related to Apache to the students who had to gather information themselves and present and discuss the results in their group. An examination at the end of the seminar was intended to check if the students could explain concepts and pieces of source code of Apache.

One result of the seminar — apart from the student's experiences and their presentation slides — is a set of diagrams and explanatory texts describing various aspects of Apache and its environment. These can be obtained from [4].

2.1. Sources of Information

The first task in the seminar was to find sources of information about Apache. Starting from the Apache HTTP Server Project Web site [1], it is easy to find information

about usage and administration of Apache; look at [2] as a good example. Finding information about the implementation of Apache aside from the source code was much harder. The best source of information was "Writing Apache Modules with Perl and C" [7]. This book describes the Apache Module API, a plug–in mechanism for server extensions, and provides the information needed to create new modules. It contains a description of the Apache API and the Request–Response–Loop, which is the main HTTP server loop where most module handlers are called from.

The remaining source of information about the implementation of Apache was the source code distribution of Apache itself. Aside from the partly documented source code, it also contains documentation of various details, but provides little information about the conceptual architecture.

The Apache source code distribution provides one source base for many system platforms and makes excessive use of preprocessor directives like `#ifdef`s and macros. When reading the code, you must always check if it will be compiled or skipped by the preprocessor and if a macro is replaced by code or a by a comment. For the seminar, we decided to study the code for the Linux platform only.

2.2. Tools and Notation

In the seminar, a simple tool was used for the analysis of the source code which transformed the C source code into a set of syntax–highlighted and hyper–linked HTML files. Now the students could navigate in the source code from any function call to its definition with a web browser. The tool has been inspired by doxygen [8] and takes care of the excessive use of preprocessor statements in the source distribution of Apache 1.3.

Further code analysis tools were not used for two reasons:

- An important amount of information needed for the conceptual architecture is not existent in the code and therefore cannot be extracted by a tool.
- Students have to learn how to structure and categorize code and how to extract information for different aspects like multitasking or communication. After having learned to do this successfully for a small product like Apache, they can use tools to examine bigger products.

In the curriculum, HPI students are taught the fundamental modeling concepts (FMC, see [6] for an introduction) during semester 1 – 3. They provide a simple but powerful terminology and notation to model both the conceptual and execution architecture view (see [9], [10] and [5]).

2.3. A systematic approach to analyzing and understanding a software product

The structure of the seminar reflects the steps you have to take for a systematic approach to analyzing a software product:

1. Defining the purpose of the analysis
2. Gathering domain knowledge and understanding the system
3. Understanding the function and handling of the software product
4. Understanding the implementation of the product (if sources are available)

In the seminar, the students had to share their knowledge with the group, so comprehensive diagrams and an adequate presentation played an important role. Finding and formulating the topics was a task the authors did prior to the seminar. In real–life situations, however, you usually have to start by defining the topics yourself.

In the following, the detailed steps and some of the topics given to the students can be found:

1. Defining the purpose of the analysis The level of detail of the following steps depend on the target of the analysis.

This goal for the seminar was: The students should be able to explain key concepts of the system in general, of Apache and its implementation. For the latter they had to be able to explain some parts of source code of the server runtime (see section 3).

2. Gathering domain knowledge and understanding the system First make a list of information sources and a glossary for domain terms. You will add more items or correct them in the following phases. Then look at the system consisting of the software product and its environment. Often you need a lot of domain knowledge to understand the purpose and the behavior of the product. It is crucial to gather information about the communication partners, the protocols used for their communication and the structures of external data sources.

The students had to understand and model the role of HTTP clients and servers, TCP/IP, DNS, the protocol HTTP/1.1, authentication, SSL, scripting, cookies, proxy, caching, virtual hosts and so on. A big help in understanding the protocols was to "talk" HTTP to the server with `telnet` to examine the response of an HTTP server and to implement and alter a simple HTTP server as shown in figure 2 to learn what a browser is able to do. The result was a model of the conceptual architecture of the entire system.

3. Understanding the function and handling of the software product Here you learn how to install, configure and administrate the product, about its features and its extendibility via APIs. This knowledge leads to a conceptual architecture model of the internal structure of the software product which might not be identical to the real runtime structure, but is sufficient to explain its behavior. To get information about details not clarified in the documentation, you either have to experiment with the product or study its implementation, if available.

The students had to compile, install and configure Apache and present the module API which reveals a lot of information about the internal structure of Apache. Furthermore they had to implement a small module to extend the behavior of Apache. This led to a conceptual architecture model of Apache which can be used to explain its features.

4. Understanding the implementation of the software product If the source code is available, you should make a table of contents of all files of the source distribution, classify them and decide which of them contain probably important information. The source distribution of Apache 1.3.17 comes in 780 files in 44 subdirectories, 235 files contain C source code. Only a handful of them are essential for understanding the runtime system structure. The structure of the code usually differs from the conceptual architecture, because it has to respect aspects like maintainability, division of labor between many developers, changeability and many more.

Using the conceptual architecture model of step 3 as a starting point, you can study the implementation to verify and enrich the model and dive into detail where you need more information about the product (this depends strongly on the purpose of the analysis). Often you need additional information about library functions or operating system calls. The conceptual architecture model serves as a map where you can find the proper place for implementation details and fill the white areas with information extracted from the code. Additionally, the model is an excellent basis for communication about the code.

The students examined how Apache starts up and shuts down, where and how it handles multitasking and concurrency. They looked at its resource management, the plugin mechanism (Apache Modules), the Apache API, the main server loops, how it collects the configuration information in order to process a request, and the dynamic loading of extension modules. In addition, they had to study operating system calls for process handling (fork, exec), signals, sockets, pipes, memory management and so on.

According to the goal defined for the seminar, the students studied only a small but important part of the code. They focused on understanding the server runtime, i.e. start–up and shutdown of the server, the maintenance loops of the master server and the request–response loop of the Child Servers, where most module handlers are called from. The CGI module served as a prototype for all other modules.

3. The conceptual Architecture of Apache

3.1. HTTP servers in general

In general, an HTTP server waits for requests and answers them according to the Hypertext Transfer Protocol. A client (usually a web browser) requests a resource (usually an HTML document or an image). The server examines the request and maps the resource identifier to a file or forwards the request to a program which then produces the requested data. Finally, the server sends the response back to the client.

Since HTTP is a stateless protocol, the server doesn't have to keep any session information for subsequent requests.

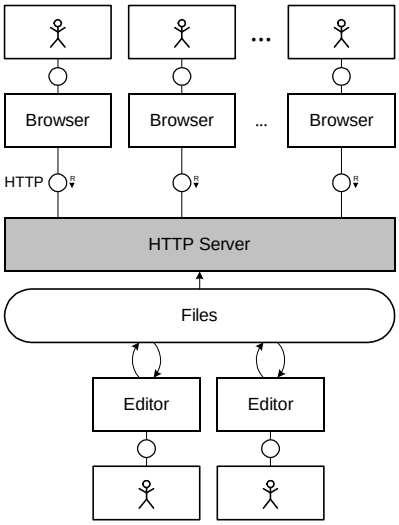

Figure 1. System structure of an HTTP Server and its environment (block diagram)

The general idea behind the World Wide Web is a system where authors provide information for readers. They use the technical infrastructure provided by HTTP servers, browsers and a network. Figure 1 shows a compositional structure[1] of the system in general. It shows one HTTP

[1]Notation of a compositional structure (block) diagram: Rectangles symbolize active components (agents) like people (symbolized by a stick man), machines or processes, big circles and ellipses stand for passive components like storages and small circles on a line depict channels between agents.

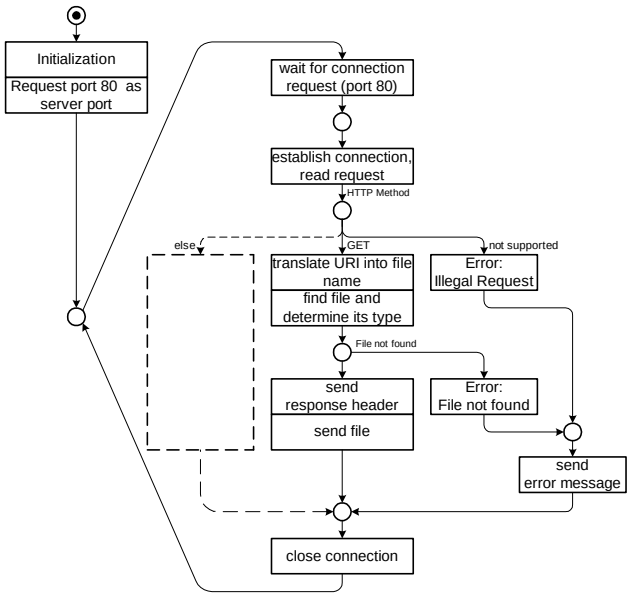

Figure 2. Behavior of a single–tasking HTTP server (Petri net)

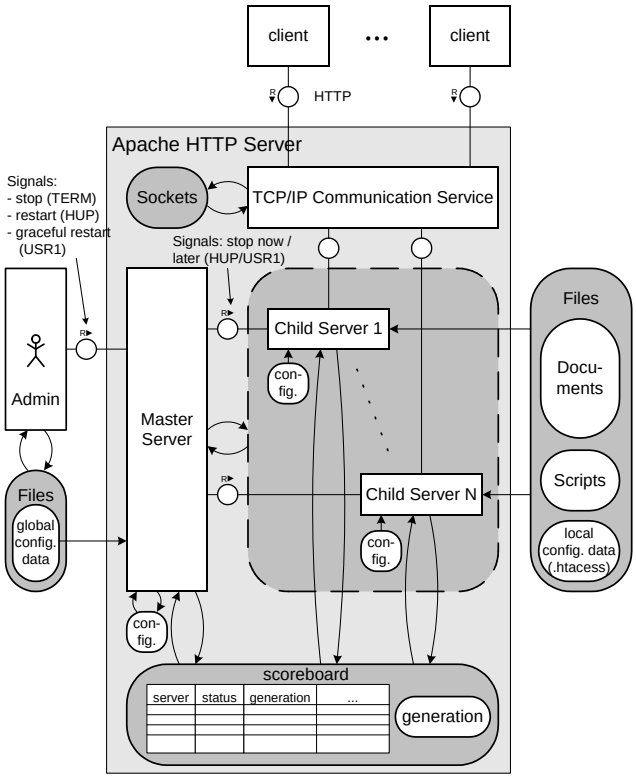

Figure 3. System structure of Apache at runtime (block diagram).

server and many clients using the communication protocol HTTP. A client consists of a Web Browser and a human being controlling it. The server can read files from a storage (file system or database) and send them in response to a request to a browser.

The behavior of a single–tasking HTTP Server is shown in figure 2. After the initialization, the server enters the request–response loop. For simplicity, only the response to a GET request is shown.

It is very easy to implement an HTTP server like that with 100 lines of code [4]. An HTTP server suitable for daily use, however, must provide additional features like serving multiple clients simultaneously, security, robustness, scripting and many more.

3.2. Conceptual Architecture of Apache

In this section, the focus lies on the conceptual architecture of Apache, its behavior during startup, shutdown and on the server maintenance loop. Further details of topics like the request processing or the module structure can be found in [4].

The conceptual architecture shown below represents a general pattern for stateless multitasking network servers.

The system structure at runtime Figure 3 shows a snapshot of the runtime structure of Apache after initialization. The environment is similar to the system view in figure 1. The administrator controls the HTTP server via signals and via configuration files. The files are partitioned into the documents (HTML files, images, applets, etc.), server–side scripts and local configuration data (.htaccess files[2]).

The inner structure of Apache shows three types of agents: The Master Server process, the TCP/IP Communication Service and a variable number of Child Server processes.

- The Child Servers are responsible for serving HTTP requests. They run the request–response loop similar to figure 2.
- The TCP/IP Communication Service is part of the operating system and manages access to TCP ports and connections. It can receive connection requests simultaneously and wake up processes waiting for a request.
- The task of the Master Server is to create and control the Child Servers and to act as the representative of the

[2] An .htaccess configuration file is stored in a document directory and can be used to apply a special configuration — usually access restriction — locally to the directory and its subdirectories. In contrast to the global configuration, one doesn't need administrator privileges to change a local configuration.

Apache Server towards the Administrator. It also reads and processes the configuration data and gives a copy of it to every Child Server during creation.

The Master Server must guarantee that there are always enough idle Child Servers ready to process incoming requests. It therefore needs to know about the state of each Child Server. Therefore it sets up the so–called scoreboard inside a shared memory area where each Child Server has to refresh its current state.

Behavior of the Apache Server and its components Figure 4 shows the general behavior of Apache concerning start–up, shutdown and the most important loops[3].

After starting Apache, only one process exists. This process does the first–time initialization, reads the configuration and then detaches itself from the shell. This results in the creation of a new process that will become the Master Server shown in figure 3.

The Master Server enters the *restart loop* and performs the master initialization. As this is the first time this loop is run, it executes the right branch in figure 4 (non–graceful): After reading the configuration for the new generation of Child Servers, it sets up a new scoreboard, starts as many new Child Server processes as defined in the configuration and adds an entry for each of them in the scoreboard. As the Master Server process uses the `fork()` system call to create a Child Server, each of them gets its own copy of the configuration data — this is modeled as small storages below each Child Server in figure 3.

Simultaneously, every Child Server now enters its *request–response loop* and starts waiting for a request. The *keep–alive loop* is a sub–loop of the request–response loop and enables the reuse of an existing TCP connection for subsequent requests from the same client[4]. As long as a Child Server runs the keep–alive loop, it can only handle requests coming from that connection! Therefore it leaves the keep–alive loop after a certain time of inactivity, usually 15 seconds.

In the meantime the Master Server enters the *master server loop* to control and maintain the Child Servers. It has to keep the number of Child Servers within a given range, and whenever a Child Server dies, it has to replace it with a new copy. The Master Server must guarantee that there are always enough idle Child Servers ready to handle a new request, but it must avoid a waste of resources by keeping too many idle Child Servers.

Whenever the Administrator forces a restart or shutdown of the Apache Server, the Master Server kills the Child Servers and either enters the *restart loop* again or cleans up and exits.

The communication between Master and Child Server is done via signals and via the scoreboard — see figure 3.

A *graceful restart* avoids the interruption of the handling of pending requests that occurs when a normal restart is initiated. In this case, the Master Server sends a special signal to the Child Servers to indicate that only idle servers should exit while the busy ones can go on and finish their job. The Master Server reads the new configuration and enters the master server loop directly without starting new Child Servers or cleaning up the scoreboard (the left branch of the master initialization in figure 4). Instead, it replaces the Child Servers that have just exited after receiving the signal, and adjusts their number in case the allowed range has been changed in the new configuration.

After handling a request, a Child Server checks in the scoreboard if its own generation matches the current generation, see figure 3 below. If not, it exits and gets replaced with a new Child Server by the Master Server.

The runtime architecture presented above guarantees quick responses to requests, because there is always a pool of idle, fully configured server processes ready to handle incoming requests.

3.3. Apache 2

The Apache Group has been developing Apache 2 for several years now. It is a rewrite of the Apache Server avoiding the source fragmentation caused by the excessive use of preprocessor directives (`#ifdefs` and macros) in Apache 1. The new Apache provides a better code structure, an extended module interface and a universal server API called Apache Portable Runtime (APR). The Multiprocessing Modules (MPM) provide a flexible way to handle multitasking dependent on operating systems and performance requirements. Now it's easier to integrate a new platform and to use a combination of processes and threads on the Unix platform.

The *Preforking MPM* provides the same conceptual architecture as Apache 1.3, shown in figure 3 and 4, while the mapping to source code files has changed. The following selection of Multiprocessing Modules of Apache 2 provide new elements for the conceptual architecture:

Worker MPM: Again, a Master Server controls the number of Child Servers, depending on the current server load. Each Child Server process is a composition of one listener thread, a job queue and a definite number

[3] The dotted lines serve as graphical comments to indicate the creation of a Child Server process resulting in a new petri net for the new Child Server. This modeling decision was made to address the problem of structure variance in petri nets.

[4] This is called "persistent connection" in HTTP/1.1 [3] and results from the fact that an HTML file is supplemented by images, style sheets or applets needed for presentation. The browser requests these files immediately after receiving and parsing the HTML file. It would be a waste of resources if it had to establish a new TCP connection for every request.

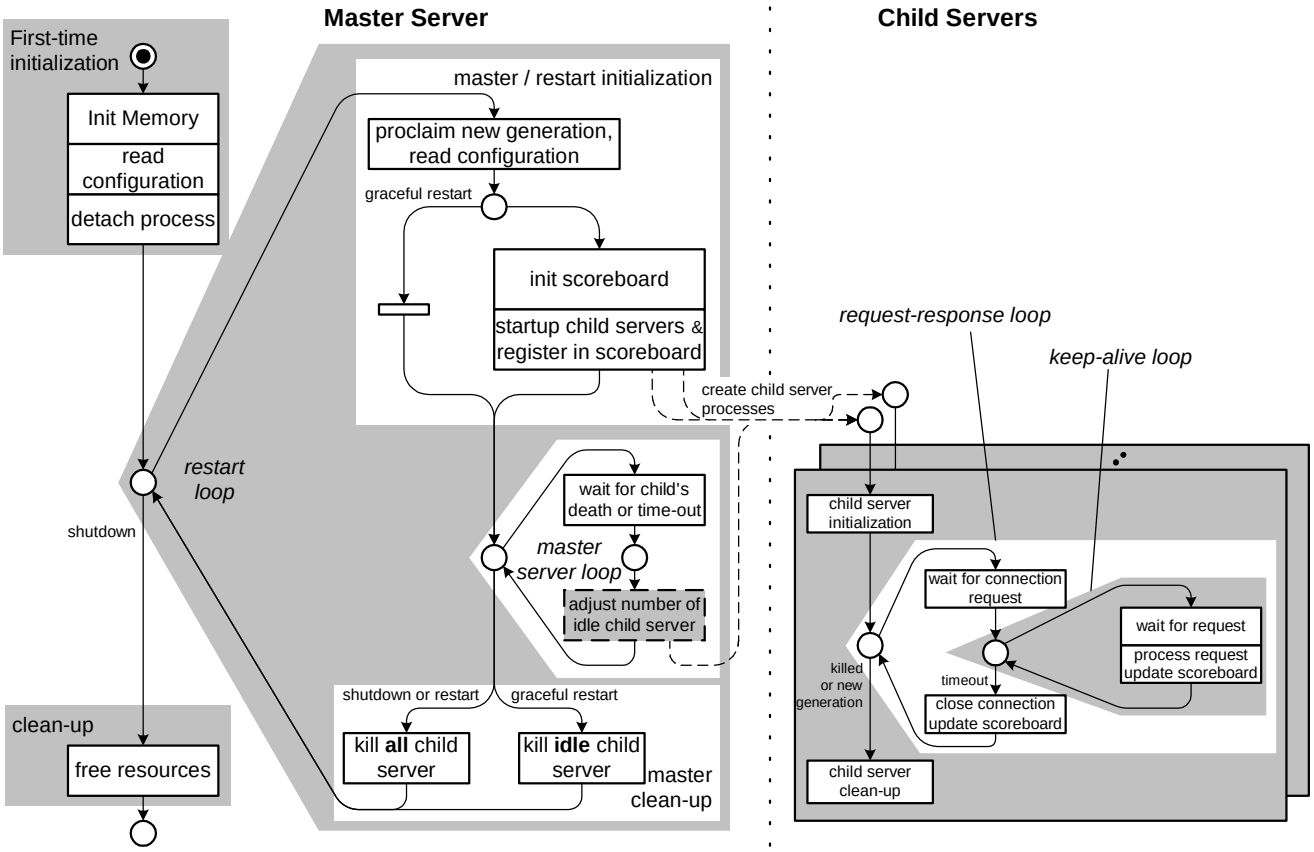

Figure 4. Overall behavior of the Apache HTTP server (petri net).

of worker threads — see figure 5. The listener waits for requests and puts them into a job queue. A job contains the socket ID for the TCP connection. A worker waits to get a job from the queue and then reads and processes the request. The exit flag is used for the graceful restart.

WinNT MPM: The server structure resembles the Worker MPM with an important difference: There is only one Child Server process, so the Master Server process just monitors this Child Server process and restarts it if it dies. The Child Server process contains a controller thread, a listener thread for each server port and a definite number of worker threads — similar to figure 5.

4. Conclusion

Domain knowledge is the most important prerequisite for a good conceptual architecture model. Although all students use the World Wide Web regularly, they needed half of the time to understand the details of the protocols and procedures and to make the necessary abstractions.

After that, the students could quickly understand the features of Apache and managed to configure and work with it, because they could use the domain knowledge and the model of the first part of the seminar. The conceptual architecture model of Apache was developed and refined with the aid of the authors. Even after the analysis, there were still many white areas on the map: For example, it is not documented well how to process new configuration directives of your own extension module for Apache. This could only be clarified after looking at the implementation of Apache[5].

Only the last third of the seminar was dedicated to the implementation of Apache. Understanding the modular design of Apache helped reducing the code for the analysis. The students learned a lot about the problems and the implementation of a multitasking server.

The high–level FMC compositional structure and behavior diagrams presented in this paper have proven to be useful to introduce the conceptual architecture of Apache to a broad audience. Their creation took time, because they had to go through many improvement and abstraction cy-

[5]The study of the configuration engine of Apache was done by Oliver Schmidt at the HPI. A publication will be available at [4].

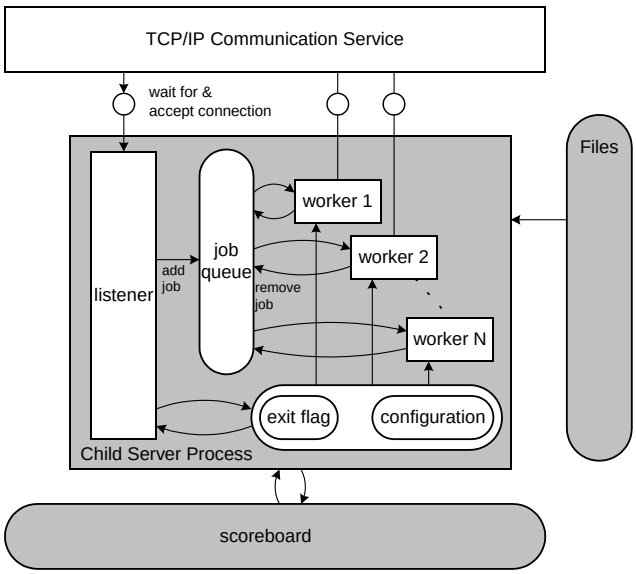

Figure 5. Apache 2: Listener and Worker threads inside a Child Server (block diagram)

cles until they were satisfactory. The improvements often concerned layout issues because these can be crucial for communicating software architecture in an efficient way.

Without a conceptual architecture model and the abstractions needed to create it, it is not possible to understand the implementation of bigger systems in a reasonable time. Finding these abstractions is supported by the FMC terminology and requires practice. Therefore we offer the seminar in 2002 with more emphasis on modeling and abstraction.

References

[1] T. Apache Software Foundation. Apache http server project. [Online] http://httpd.apache.org.

[2] R. S. Engelschall. *Apache Desktop Reference*. Addison–Wesley, 2000.

[3] Fielding et al. Hypertext transfer protocol — http/1.1 rfc 2616, 1999.

[4] B. Gröne, A. Knöpfel, and R. Kugel. The apache modelling project. [Online] http://www.hpi.uni-potsdam.de/apache, 2002.

[5] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison–Wesley, 1999.

[6] F. Keller, P. Tabeling, et al. Improving knowledge transfer on architectural level: Concepts and notations. In *(to appear) The 2002 International Conference on Software Engineering Research and Practice*, 2002.

[7] L. Stein and D. MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly, 1999.

[8] D. van Heesch. Doxygen – a documentation system for c++, java, idl and c. [Online] http://www.doxygen.org, 2002.

[9] S. Wendt. Einführung in die begriffswelt allgemeiner netzsysteme. *Regelungstechnik*, 30(1), 1982.

[10] S. Wendt. *Nichtphysikalische Grundlagen der Informationstechnik*. Springer Verlag Heidelberg, 2nd edition, 1991.

The realâ€“life system examined in the course was the World Wide Web (WWW) and the Apache HTTP Server. The conceptual architecture of the system was modeled using the Fundamental Modeling Concepts (FMC) which turned out to be well suited for sharing knowledge about both concepts and details of the system. Excerpts of the model are presented in this document.Â @inproceedings{Grne2002ArchitectureRO, title={Architecture recovery of Apache 1 . 3 â€” A case study}, author={Bernhard Gr{\"o}ne and Andreas Kn{\"o}pfel and Rudolf Kugel and Hasso Plattner}, year={2002} }. Bernhard GrÃ¶ne, Andreas KnÃ¶pfel, +1 author Hasso Plattner. Published 2002.