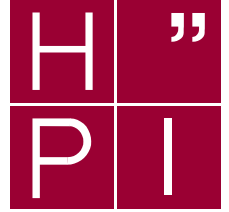




HASSO-PLATTNER-INSTITUT
für Softwaresystemtechnik an der Universität Potsdam



The Apache Modeling Project

Bernhard Gröne, Andreas Knöpfel,
Rudolf Kugel und Oliver Schmidt

Technische Berichte
des Hasso-Plattner-Instituts
für Softwaresystemtechnik an der Universität Potsdam

The Apache Modeling Project

Bernhard Gröne, Andreas Knöpfel,
Rudolf Kugel und Oliver Schmidt

Juli 2004

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Die Reihe *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam* erscheint aperiodisch.

Herausgeber: Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Redaktion Bernhard Gröne, Sabine Wagner
EMail bernhard.groene@hpi.uni-potsdam.de

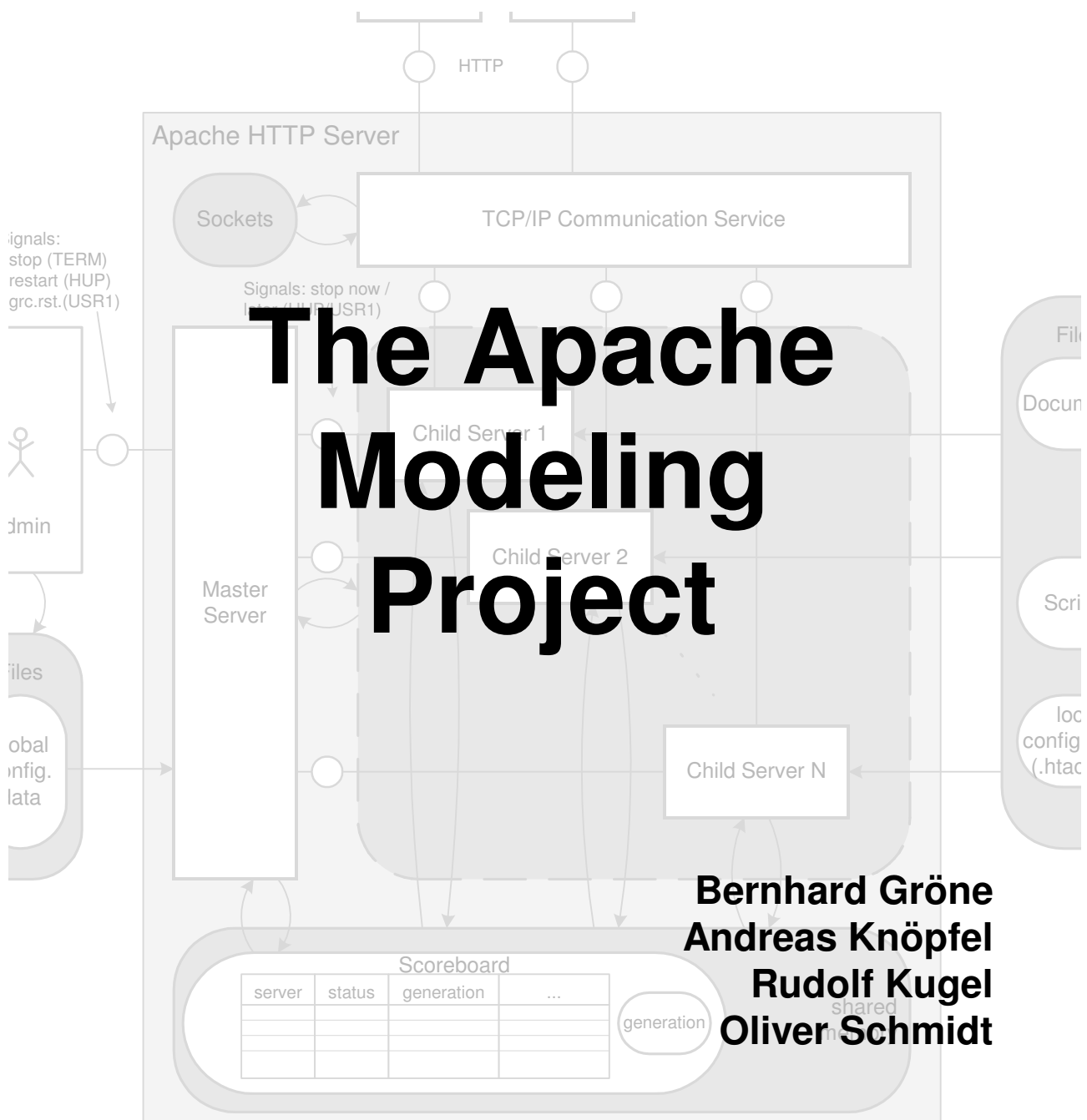
Vertrieb: Universitätsverlag Potsdam
Postfach 60 15 53
14415 Potsdam
Fon +49 (0) 331 977 4517
Fax +49 (0) 331 977 4625
e-mail: ubpub@rz.uni-potsdam.de
<http://info.ub.uni-potsdam.de/verlag.htm>

Druck allprintmedia gmbH
Blomberger Weg 6a
13437 Berlin
email: info@allprint-media.de

© Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, 2004

Dieses Manuskript ist urheberrechtlich geschützt. Es darf ohne
vorherige Genehmigung der Herausgeber nicht vervielfältigt werden.

Heft 5 (2004)
ISBN 9-937786-14-7
ISSN 1613-5652



The Apache Modeling Project

Bernhard Gröne
Andreas Knöpfel
Rudolf Kugel
Oliver Schmidt



July 8, 2004

Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, Oliver Schmidt:

The Apache Modeling Project



Hasso-Plattner-Institute for Software Systems Engineering

Prof.-Dr.-Helmert-Straße 2-3, D-14482 Potsdam

P.O.Box 90 04 60, D-14440 Potsdam, Germany

<http://www.hpi.uni-potsdam.de>

The current version of this document can be found at the HPI Apache Modeling Project web site:

HTML: <http://apache.hpi.uni-potsdam.de/>

PDF: http://apache.hpi.uni-potsdam.de/modules.php?name=Downloads&d_op=getit&lid=3

Thanks to:

Robert Mitschke and Ralf Schliehe-Diecks for preparing and editing material for this document.

Copyright (c) 2002–2004 Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, Oliver Schmidt. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in appendix D entitled "GNU Free Documentation License".

Abstract

This document presents an introduction to the Apache HTTP Server, covering both an overview and implementation details. It presents results of the Apache Modelling Project done by research assistants and students of the Hasso-Plattner-Institute in 2001, 2002 and 2003. The Apache HTTP Server was used to introduce students to the application of the modeling technique FMC, a method that supports transporting knowledge about complex systems in the domain of information processing (software and hardware as well).

After an introduction to HTTP servers in general, we will focus on protocols and web technology. Then we will discuss Apache, its operational environment and its extension capabilities — the module API. Finally we will guide the reader through parts of the Apache source code and explain the most important pieces.

Contents

1	Introduction	1
1.1	About this document	1
1.2	The FMC Philosophy	1
1.3	The modeling project	2
1.4	Sources of Information	2
2	HTTP Servers	4
2.1	Introduction	4
2.2	Tasks of an HTTP Server	4
2.3	Protocols and Standards	7
2.3.1	RFCs and other standardization documents	7
2.3.2	TCP/IP	7
2.3.3	Domain Name Service (DNS)	9
2.3.4	HTTP	11
2.4	Access Control and Security	17
2.4.1	Authorization	17
2.4.2	Authentication methods	18
2.4.3	HTTPS and SSL	21
2.5	Session Management	23
2.5.1	HTTP — a stateless protocol	23
2.5.2	Keep the state	23
2.6	Dynamic Content	24
2.6.1	Server-side Scripting	25
3	The Apache HTTP Server	27
3.1	Overview	27
3.1.1	History	27
3.1.2	Features	28

3.2	Using Apache	29
3.2.1	Configuration	29
3.2.2	Performance	35
3.3	Extending Apache: Apache Modules	35
3.3.1	Introduction	35
3.3.2	Types of Handlers	37
3.3.3	Content Handling	39
3.3.4	Apache 2 Filters	40
3.3.5	Predefined Hooks	41
3.3.6	Inside a Module: mod_cgi	46
3.3.7	The Apache API	49
4	Inside Apache	53
4.1	Introduction	53
4.2	Structure of the Source Distribution	54
4.2.1	Apache 1.3.17 source distribution	54
4.2.2	Apache 2.0.45 source distribution	55
4.3	Multitasking server architectures	55
4.3.1	Inetd: A common multitasking architecture	56
4.3.2	Overview — Apache Multitasking Architectures	58
4.3.3	The Preforking Multiprocessing Architecture	58
4.3.4	Apache Multitasking Architectures and MPMs	73
4.3.5	Win32/WinNT MPM	75
4.3.6	Worker MPM	78
4.3.7	Others MPMs	79
4.4	The Request–Response Loop	80
4.4.1	Overview	80
4.4.2	Waiting for connection requests	80
4.4.3	Waiting for and reading HTTP requests	82
4.4.4	Process HTTP Request	82
4.5	The Configuration Processor	85
4.5.1	Where and when Apache reads configuration	85
4.5.2	Internal data structures	87
4.5.3	Processing global configuration data at start–up	89
4.5.4	Processing configuration data on requests	94



4.6	Memory and resource management: Apache Pools	98
4.6.1	Why another memory management	98
4.6.2	The pool structure	99
4.6.3	The pool API	100
A	Operating System Concepts	102
A.1	Unix Processes	102
A.1.1	fork()	103
A.1.2	exec()	104
A.1.3	wait() & waitpid()	105
A.1.4	kill()	105
A.2	Signals and Alarms	105
A.2.1	Signals	105
A.2.2	Usage	106
A.3	Sockets	106
A.3.1	TCP vs. UDP Ports	107
A.3.2	Asynchronous Events	107
A.3.3	Sockets in general	107
A.3.4	TCP Sockets	108
A.3.5	UDP Sockets	110
A.4	Pipes	110
A.4.1	Using pipes for inter process communication	110
A.4.2	STDIN, STDOUT, STDERR	111
A.4.3	Implementation	111
A.4.4	Named pipes	111
A.5	Longjmp	112
A.5.1	Concept	112
A.5.2	Example Longjump	113
A.5.3	Common application	115
A.5.4	Common Pitfalls	116
B	Sources	118
B.1	Simple HTTP Server	118

C	Fundamental Modeling Concepts (FMC)	121
C.1	What is FMC?	121
C.2	Compositional structures and block diagrams	122
C.2.1	Example system	122
C.2.2	Block diagrams	122
C.3	Dynamic structures and Petri nets	123
C.3.1	Behavior of dynamic systems	123
C.3.2	Petri nets	125
C.4	Value range structures and entity relationship diagrams	126
C.4.1	Structured values and their relationships	126
C.4.2	Entity Relationship Diagrams	126
C.5	Levels of abstraction	128
C.5.1	High-level structures	128
C.5.2	Hierarchy of models	128
C.5.3	Lower-level structures	128
C.6	What is special about FMC?	130
D	GNU Free Documentation License	131
	Bibliography	135
	Glossary	136
	Index	139

List of Figures

2.1	Simple system (structure and behavior)	5
2.2	Behavior of a simple HTTP server	6
2.3	Establishing and finishing a TCP connection	8
2.4	DNS Zones	10
2.5	Example of a HTTP Request/Response message pair	12
2.6	Value structure of a HTTP Request/Response (overview)	12
2.7	Value structure of a HTTP Request/Response (details)	13
2.8	Basic Authentication process	19
3.1	The Apache HTTP Server in its environment	28
3.2	Configuring the Apache HTTP Server via configuration files	30
3.3	Apache 2 Module structure	36
3.4	Interaction of Core and Modules	36
3.5	Apache 2.0 hook mechanism	38
3.6	Apache Filters: The Input / Output filter chain	40
3.7	Apache Filters: A Brigade contains a series of buckets	41
3.8	Apache Filters: A Filter chain using Brigades for data transport	41
3.9	Apache request–response loop with module callbacks	44
3.10	Apache request processing with module callbacks	45
4.1	Directory structure of the Apache HTTP Server 1.3.17 source distribution	54
4.2	Directory structure of the Apache HTTP Server 2.0.45 source distribution	55
4.3	Multiprocessing Architecture of an inetd server	56
4.4	Behavior of a multiprocessing server	57
4.5	The leader–followers pattern used in the preforking server architecture	59
4.6	The Apache 2.0 Preforking MPM	60
4.7	Overview: The behavior of Apache	61
4.8	Details of the behavior of Apache	63

4.9	The Master Server Loop	68
4.10	Responsibility of an Apache 2.0 MPM	74
4.11	The Apache 2.0 WinNT MPM	76
4.12	The Apache 2.0 Worker MPM	78
4.13	The Request–Response Loop	81
4.14	Processing of a request	83
4.15	Configuration processing components of Apache	85
4.16	Points in time when Apache reads configuration	86
4.17	Apache data structures concerning configuration	87
4.18	Apache configuration data structures in detail	88
4.19	Structure of the configuration processor (focusing on the data flow)	90
4.20	Reading configuration data: Layering of function calls	91
4.21	Processing configuration files with section directives	93
4.22	Structure of the per-request configuration processor: The walk procedures	95
4.23	The directory walk (without error handling)	96
4.24	Hierarchy of built-in Apache Pools	99
4.25	Linked List of Blocks	100
A.1	Multiprocessing and Multithreading	102
A.2	Internals of the fork() system call	104
A.3	Basic concept of sockets	107
A.4	Communication via sockets	108
A.5	Dialup and connection sockets	109
A.6	Sockets and ports	109
A.7	Pipes	112
A.8	longjmp behaviour	114
A.9	Stack content before calling longjmp() in the example above	115
A.10	Calling longjmp() can cause errors when used without caution	116
C.1	Block diagram: Travel agency system	122
C.2	Petri net: Buying a ticket	124
C.3	Entity relationship diagram: Tour reservations	126
C.4	Block diagram: Travel agency system	129
D.1	FMC Block Diagrams (Part 1: Example)	143
D.2	FMC Block Diagrams (Part 2: Description)	144



D.3 FMC Entity/Relationships Diagrams (Part 1: Example)	145
D.4 FMC Entity/Relationships Diagrams (Part 2: Description)	146
D.5 FMC Petri Nets (Part 1: Basic components)	147
D.6 FMC Petri Nets (Part 2: Basic Example)	148
D.7 FMC Petri Nets (Part 3: Advanced features)	149
D.8 FMC Petri Nets (Part 4: Advanced Example)	150

Chapter 1

Introduction

1.1 About this document

This document is intended for everybody interested in technical aspects of HTTP Servers, especially Apache. It does not present much information about usage and administration of Apache — this is covered by many other documents. We try to give both an overview as well as dive into some details necessary for understanding how Apache works and how it is constructed.

Chapter 2 shows the tasks of an HTTP server in general and its behavior in its environment. We focus on protocols like HTTP and issues concerning security, session management or dynamic content. Chapter 3 introduces the Apache HTTP Server, its features, configuration, administration and extension API.

Chapter 4 guides the reader through the implementation details of Apache. It is intended for developers and people who want to know how to implement a multitasking network server. It introduces concepts and their implementation in the Apache source code. The chapter is far from being complete but tries to show the most important concepts needed to understand the Apache source code.

All diagrams in this document follow the notation of the Fundamental Modeling Concepts (FMC). Appendix C presents a short introduction to this modeling method, covering philosophy, concepts and notation.

1.2 The FMC Philosophy

Analyzing and understanding existing software is a common task in software engineering. As software systems tend to become more complex, there is the need for division of labor. People working on a software product must share their knowledge — the team needs to communicate about the software. Deficiencies become most apparent whenever a new team member has to gather information about the software product or the team manager is not directly involved in the development and has to keep track of what is going on.

In our group “Modeling software intensive systems” at the Hasso-Plattner-Institute for Software Systems Engineering, we focus on efficient communication and sharing of knowledge about software systems.



A *model* is an abstraction of the software system that helps us to think and talk about it. Everyone dealing with a software systems forms a unique model of the system in his or her mind. Division of labor works best if those models resemble each other.

The Fundamental Modeling Concepts (FMC) help in transporting a model of a software system from one human being to another. That is the primary objective. Visit our web site at <http://fmc.hpi.uni-potsdam.de> to learn more about FMC.

1.3 The modeling project

The project was done at the Hasso–Plattner–Institute for Software Systems Engineering. The idea behind the project was to show students a way to master the complexity of a real software product. Apache is a HTTP Server developed by many volunteers and used by many companies all over the world. Most important, its source code is available. Therefore it is an interesting object for a modeling project. We soon learned that a lot of information about configuration and administration of Apache exists, but little about implementation details aside from the source code.

More details of the seminars of 2001 and 2002 can be found at the Apache Modeling Portal (<http://apache.hpi.uni-potsdam.de> [3])

A result of the project was a set of models describing various aspects of Apache and its environment which formed the basis of this document.

1.4 Sources of Information

The first task in the modeling project was to find sources of information about Apache. Starting from the Apache HTTP Server Project Web site (<http://httpd.apache.org> [2]), it is easy to find information about usage and administration of Apache; take for example “Apache Desktop Reference” by Ralf S. Engelschall [1]. Finding information about the implementation of Apache was much harder. A very good source of information was “Writing Apache Modules with Perl and C” by Lincoln Stein & Doug MacEachern [4]. This book describes the Module API of Apache and provides the information needed to create new modules. Therefore it contains a description of the Apache API and the Request–Response–Loop (see section 4.4).

The most important source of information was the source code distribution of Apache itself. It contains documentation of various details and lots of comments in the source code. One problem lies in the fact that the source code distribution provides one source base for many system platforms. In the Apache 1.3.x source, big parts of code are only valid for one platform (using the `#ifdef` preprocessor directive). For Apache 1.3.x, we only looked at the code valid for the linux platform. In 2002, we examined the Apache 2.0 source code which provides a much better structure. The flexibility of the new Apache holds other obstacles for the reader of its code: For example, the hook mechanism provides an extensible indirect procedure call mechanism using an elegant macro mechanism. Without knowing this mechanism, you won’t find out which handlers will be registered for a hook (see section 3.3.2 for details).

The only tool used for the analysis of the source code transformed the C source code into a set of syntax highlighted and hyperlinked HTML files. They are available at our project site¹.

¹ Start with “Structure of the Apache HTTP Server source distribution”:
<http://apache.hpi.uni-potsdam.de/document/sources/>

Chapter 2

HTTP Servers

2.1 Introduction

If you want to understand a software product, it is a good idea to know exactly what it is supposed to do. In case of a server, you need information about the clients, resources and protocols, too. As the server is part of a system, it is necessary to examine and understand the system.

As a consequence, we will first give an introduction to HTTP servers in general, including protocols like HTTP, TCP/IP and finally discussing subjects like dynamic content and web applications.

Section 2.2 gives an overview of the system consisting of the HTTP Server, clients and resources. It presents a model of a simple HTTP server that could be implemented in a small exercise. Section 2.3 deals with protocols and standards related to the HTTP server. The most important protocol is HTTP. All further sections describe various aspects of the system like authentication, security or using HTTP for web applications. They just provide an overview as there are enough documents going into detail.

2.2 Tasks of an HTTP Server

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

The left-hand side of figure 2.1 shows a very simple system structure: The user communicates with the browser and the browser sends a request (R) to the HTTP server which, in turn, reads files from a storage (please note that the server has read-only access to the file storage).

This is a simplified view: The browser is a program running on a computer connected to the HTTP Server via a network. The channel between the user and the browser involves screen, mouse and keyboard. Finally, only one connection is visible although a browser can connect to multiple servers.

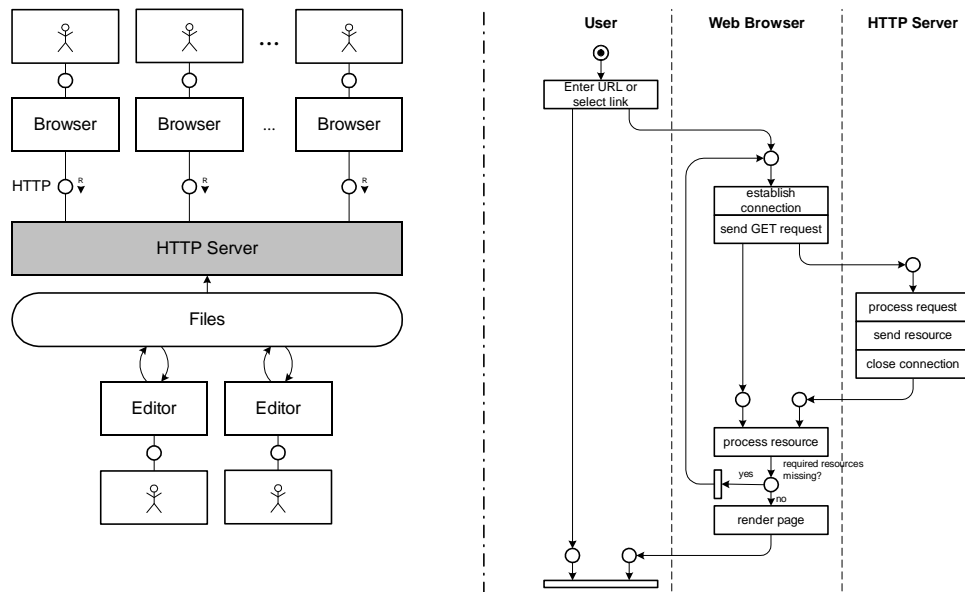


Figure 2.1: Simple system (structure and behavior)

On the right-hand side, figure 2.1 also depicts what happens in the system: The user enters a URL in the browser or clicks on a hyperlink. The Web Browser extracts the server address from the URL or link and establishes a TCP/IP connection with the server. Using the connection, the browser sends a GET request to the HTTP server. It extracts the resource name to be requested from the URL or the link.

The HTTP Server reads the request and processes it (i.e. translates it to a file name). It sends the resource in its response using the connection and finally closes the connection. The browser examines the received data. An HTML document can contain links to further resources needed to display it properly, like images or java applets. The browser has to request them from the HTTP server, too, as they are not included in the HTML document. After receiving the last resource needed to display the HTML document, the browser successfully renders the page and waits for further activities of the user.

Figure 2.1 shows a very simplified view of the system: It is easy to see that establishing and closing TCP/IP connections for each resource is a waste of time and network bandwidth. HTTP/1.0 offered the “keep alive” header field as an option, in HTTP/1.1 the default is to keep the connection alive for a certain time of inactivity of the client (usually 15 seconds). We also simplified the behavior of the browser: Since Netscape’s browser replaced Mosaic, a browser usually reacts to its user’s activities while loading images.

The dynamic structure diagram in figure 2.2 shows the behavior of the server in more detail: After initialization, the server enters the request–response loop. It waits for an incoming request, examines it¹, maps the resource to a file and delivers it to the browser. Finally it closes the connection to the client.

It is easy to implement an HTTP server displaying the behavior in figure 2.2 in about 100 lines of code (see appendix B.1 for the code). A productive HTTP server is far more complex for various reasons:

Additional Features

¹only the processing of the GET method is shown in detail.

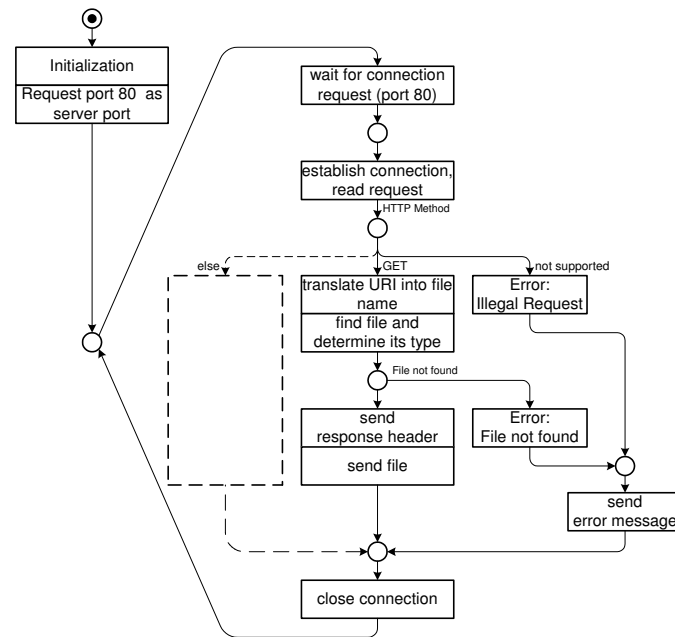


Figure 2.2: Behavior of a simple HTTP server

- full implementation of HTTP (all versions)
- handling of concurrent requests (multiprocessing / multi-threading)
- provide interfaces enabling programmers to add more features
- security (access control, authentication methods)
- dynamic web content (CGI, scripting and server-side includes)
- virtual hosts
- proxy functionality
- advanced detection of the MIME type of a resource

Operation of the server

- robust and safe operation (24 x 7)
- configuration by text files
- apply configuration changes without stopping the server
- administration tool to control the server
- status information provided by the server
- logging
- documentation for administrators

Portability and installation procedure

- one source distribution for various platforms
- little configuration effort to compile for a specific platform
- easy installation with sensible default configuration and useful templates
- documentation for developers aside from the source code

2.3 Protocols and Standards

2.3.1 RFCs and other standardization documents

RFCs (Request For Comments) are a collection of notes about the Internet which started in 1969. These notes describe many standards concerning computer communication, networking protocols, procedures, programs and concepts. All Internet standards are defined in the RFCs.

To become a standard, an RFC has to traverse three steps called Internet Standards Track:

1. It firstly is introduced as a *Proposed Standard* by the IESG (Internet Engineering Steering Group).
2. Afterwards, it can become a *Draft Standard* which is nearly a standard with only minor changes for special cases to come.
3. Finally, if it is widely used and considered to be useful for the community, it is raised to the rank of a *Standard*.

The *RFC Editor* is responsible for the final document. The RFC Editor function is funded by the Internet Society <http://www.isoc.org>.

For further information on RFCs look at <http://www.faqs.org/rfcs>. This site also contains links to all involved Internet Related Standard Organizations, like the W3C (<http://www.w3c.org>).

2.3.2 TCP/IP

TCP/IP is the most commonly used network protocol worldwide and all nodes connected to the Internet use it. TCP/IP consists of the 3 main protocols TCP (Transmission Control Protocol), UDP (User Data Protocol) and IP (Internet Protocol). UDP is a less important protocol using the lower-level Protocol IP as well. For more details, have a look at “Computer Networks” by Andrew Tanenbaum [6].



TCP and UDP

TCP and UDP are transmission protocols that use IP to transmit their data. While IP is responsible for transmitting packets to the destination at best effort, TCP and UDP are used to prepare data for sending by splitting them in packets.

TCP (Transmission Control Protocol) provides a connection for bi-directional communication between two partners using two data streams. It is therefore called a connection-oriented protocol. Before sending or receiving any data, TCP has to establish a connection channel with the target node. To provide the channel for the two data streams it has to split the data into packets and ensure that packets arrive without error and are unpacked in the proper order. That way an application using TCP does not have to take precautions for corrupted data transfer. TCP will make sure data transfer is completed successfully or report an error otherwise.

UDP (User Data Protocol) on the other hand is a much simpler technique for delivering data packets. It just adds a header to the data and sends them to its destination, regardless whether that node exists or expects data. UDP does not guarantee that packets arrive, nor does it ensure they arrive in the order they were sent. If packets are transmitted between two networks using different paths they can arrive in a wrong order. It's the application that has to take care of that. However, for applications needing fast transfer without overhead for data that is still usable even if single packets are missing or not in order, UDP is the protocol in choice. Most voice and video streaming applications therefore use UDP.

Establishing TCP Connections

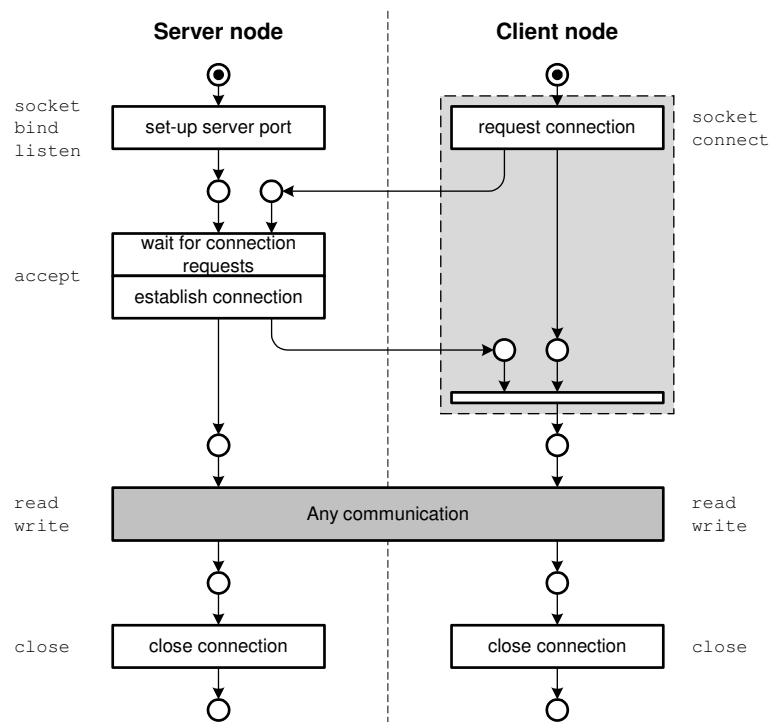


Figure 2.3: Establishing and finishing a TCP connection .

A TCP connection can only be established between two nodes: A client node sending a connection request and a server node waiting for such connection requests. After receiving a connection request, the server will respond and establish the connection. Then both nodes can send and receive data through the connection, depending on the application protocol. When finished, any node (but usually the client) can close the connection. This behavior is shown in figure 2.3. Here you also see the operating system calls used to control the sockets — see appendix A.3 for details.

Ports

An address of a TCP or UDP service consists of the IP address of the machine and a port number. These ports enable hosts using TCP/IP to offer different services at one time and to enable clients to maintain more than one connection to one single server. On a server, ports are used to distinguish services. HTTP servers usually use the well-known port 80 to offer their services. Other standard ports are 53 for DNS and 21 for FTP for example. In any situation, every connection on a network has different pairs of target and source addresses (IP address + port number).

IP addresses

IP, the Internet Protocol, is responsible for sending single packets to their destination nodes. This is accomplished by assigning each computer a different IP address. Each IP address consists of 32 bits usually represented in 4 dotted decimals each ranging from 0 through 255. An example for a valid IP address is 123.123.123.123. IP addresses can be distinguished by the networks they belong to. The IP name-space is separated into networks by dividing the 32 Bits of the address into network and host address bits. This information is used for routing the packets to its destination.

2.3.3 Domain Name Service (DNS)

As covered by the previous chapter, each node on the Internet can be identified by a unique IP address. Unfortunately IP addresses are numbers which are neither user friendly nor intuitive.

Name-space

As a solution to that problem, DNS maps user-friendly names to IP addresses. Names used in the DNS are organized in a hierarchical name-space. The name space is split up into domains. The topmost domain is the . (Dot) domain. Domains below that, referred to as first-level domains, split up the name-space by country. The first-level domains `com`, `net`, `org` and `edu` are an exception to that rule. Originally they were intended to be used in the United States only, but now are used all over the world. More first level domains will be available. Individuals can register second-level domains within almost any of these domains.

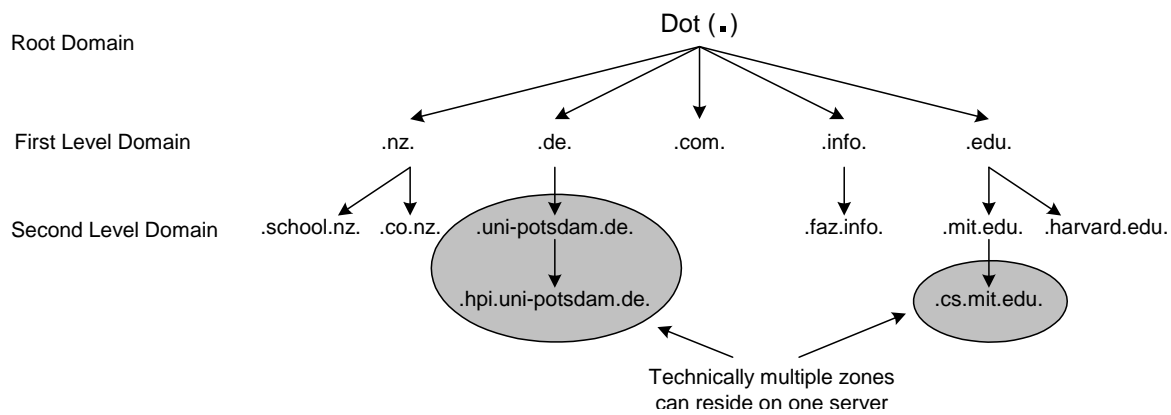


Figure 2.4: Hierarchical Structure of the DNS Namespace .

Hierarchical Server Structure

DNS Servers are organized hierarchically according to their responsibilities, forming a globally distributed database. Each DNS Server is an authority for one or multiple zones. Each zone can contain one branch of the name-space tree. A zone itself can delegate sub-zones to different name servers. The root name servers are responsible for the 'Dot' domain and delegate a zone for each first-level domain to the name servers of the corresponding country domains. These on the other hand delegate zones for each name registered to name servers supplied by or for the parties that own the second level domains. These name servers contain entries for sub-zones and/or host-names for that zone. Figure 2.4 shows zones and their dependencies.

Iterative and recursive name lookups

DNS can be queried using either recursive or iterative lookup requests. When using an *iterative* request, a DNS server will return either

1. the IP address queried,
2. a name of a DNS server which is able to successfully return the address,
3. another server's name that knows which server is closer to the name in question within the name-space tree,
4. or an error. All that information is based on the server's own database without the help of other DNS servers.

In a *recursive* request the server has to look up the IP mapped to the name provided at all cost. If the server is not responsible for the zone, it has to find out by using iterative requests with other servers. If it does not have the information in question, it will first query the root name server for the top level domain. It then will have to query the name servers that he subsequently is referred to, until one server can successfully answer the request. If no server can answer the request the last one will report an error, which will then be handed to the client that the recursive request came from. Most DNS servers responsible of the root or first level zones will only reply to iterative requests. Figure 2.4 shows the traversal of a recursive and subsequent iterative requests through the DNS server hierarchy.

2.3.4 HTTP

HTTP is the primary transfer protocol used in the World Wide Web. The first version of HTTP that was widely used was version 1.0. After the Internet began to expand rapidly, deficiencies of the first version became apparent. HTTP 1.1, the version used today, addressed these issues and extended the first version. Although HTTP doesn't set up a session (stateless protocol) and forms a simple request-response message protocol, it uses connections provided by TCP/IP as transport protocol for its reliability. HTTP is designed for and typically used in a client-server environment.

With HTTP, each information item available is addressed by a URI (Uniform Resource Identifier), which is an address used to retrieve the information. Even though URIs and URLs historically were different concepts, they are now synonymously used to identify information resources. URL is the more widely used term. An example for a URL is: `http://apache.hpi.uni-potsdam.de/index.php`. It would result in the following request

```
GET /index.php HTTP/1.1
HOST: apache.hpi.uni-potsdam.de
```

In this example the *Request URI* as seen by the web server is `/index.php`.

HTTP data structure

HTTP data transfer is based on messages. A request from a client as well as a response from a server is encoded in a message. Each HTTP message consists of a message header and can contain a message body.

An HTTP header can be split up into 4 parts.

1. Request / Status line (depending on whether it is a request or a response)
2. General Header
3. Request / Response Header
4. Entity Header

Header and body of an HTTP message are always separated by a blank line. Most header fields are not mandatory. The simplest request will only require the request line and, since HTTP 1.1, the general header field "HOST" (see section 2.3.4). The simplest response only contains the status line.

An example request/response message pair is shown in figure 2.5. The E/R diagrams in figures 2.6 and 2.7 show more details of the structure of the HTTP messages..

The next sections cover aspects of HTTP including their header fields. Important header fields not covered later are:



Request Message

```
POST /cgi-bin/form.cgi HTTP/1.1
Host: www.myserver.com
Accept: */*
User-Agent: Mozilla/4.0
Content-type: application/x-www-form-urlencoded
Content-length: 25

NAME=Smith&ADDRESS=Berlin
```

Header: Request line

Header: General

Header: Request

Header: Entity

Blank line

Body (Entity)

Response Message

```
HTTP/1.1 200 OK
Date: Mon, 19 May 2002 12:22:41 GMT
Server: Apache 2.0.45
Content-type: text/html
Content-length: 2035

<html>
<head>..</head>
<body>..</body>
</html>
```

Header: Status line

Header: General

Header: Response

Header: Entity

Blank line

Body (Entity)

␣: CR LF (Carriage Return (0x0d) + Line Feed (0x0a))

Figure 2.5: Example of a HTTP Request/Response message pair

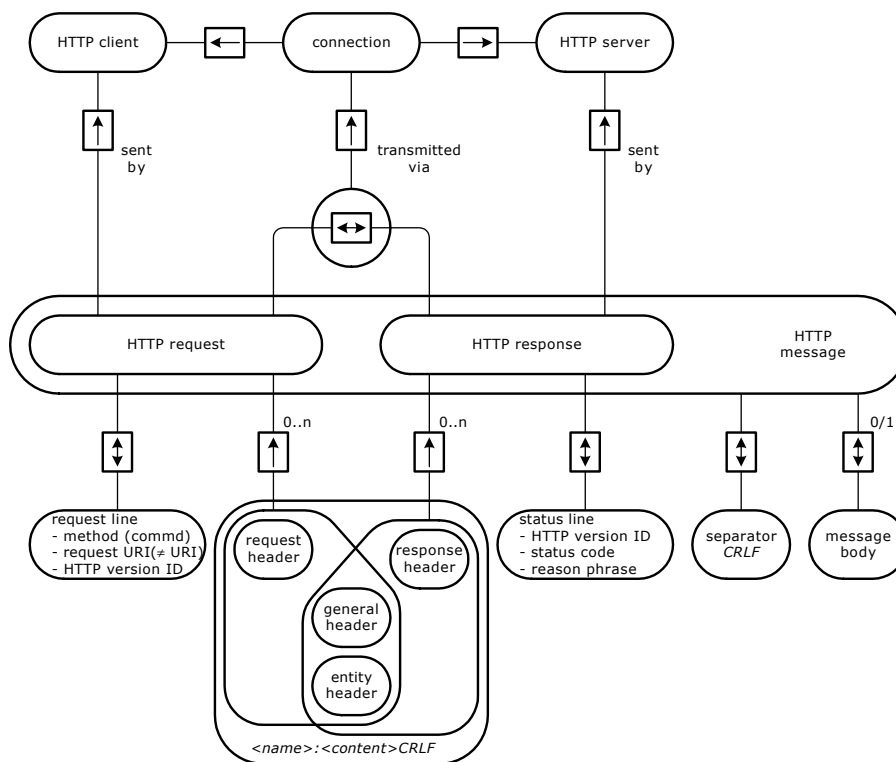


Figure 2.6: Value structure of a HTTP Request/Response (overview)

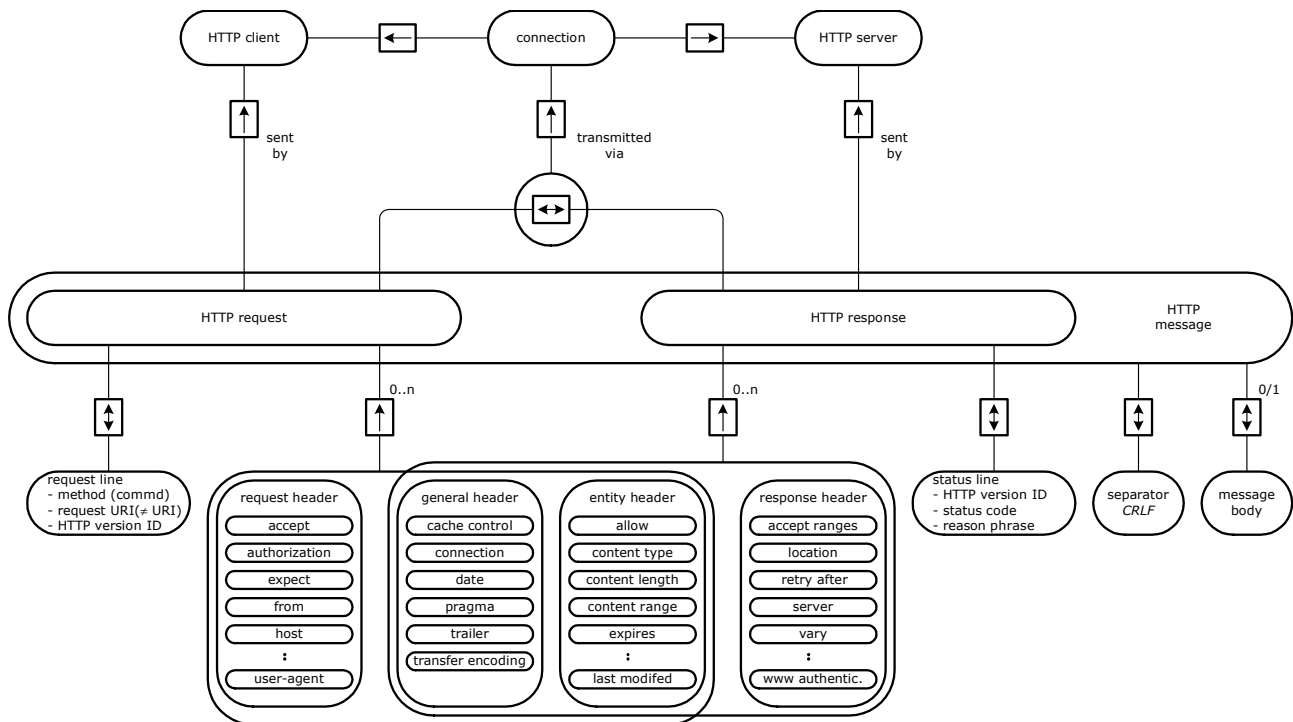


Figure 2.7: Value structure of a HTTP Request/Response (details)

- "Content-length" / "Content-type" are fields to specify the length and the MIME type of the information enclosed in the body. Any request or response including a message body uses these header fields.
- "Referer" is a field used by client applications to indicate which document referred the user to the document currently requested. Information submitted here could be stored at the server for further analysis by the webmaster.

HTTP Methods

HTTP methods are similar to commands given to an application. Depending on the method used in the request, the server's response will vary. Successful responses to some request methods do not even contain body data.

The HTTP/1.1 standard defines the methods GET, POST, OPTIONS, HEAD, TRACE, PUT, DELETE, CONNECT. The most often used methods are GET and POST.

- | | |
|------|--|
| GET | is used to retrieve an entity of information without the need to submit additional data in the message body. Before HTTP 1.0, GET was the only method to request information. |
| POST | is similar to a GET request, but POST always includes a message body in the request message to send information of any type to the server. Usually information submitted via POST is used to generate dynamic content, for further processing, or the information is simply stored to be used by other applications. POST is a method that was introduced with HTTP version 1.0. |



To send information to the server with `GET`, the client has to append it to the request URI. That causes several difficulties however:

- The length of the request URI can cause problems at the server,
- some clients can only transmit a Request URI of a certain length
- most clients display the additional Request URI information to the user

Even though `POST` is the better way to transmit additional information to the server, some applications use `GET` for that purpose, especially for small amounts of data or to allow book-marking of the URL.

All other methods are rarely used and will only be covered briefly:

HEAD	This method asks for the header of a reply only. It can be used when checking for the existence of a certain document on a web server. The response will look exactly as if requested via <code>GET</code> but will not include the message body
OPTIONS	Using this method a client can query a server for the available methods and options concerning a resource.
TRACE	The <code>TRACE</code> method is similar to ping in TCP/IP. The request message contains a mandatory header field called <code>Max-Forwards</code> . Each time the message passes a proxy server, the value of that field is decremented by one. The server that gets the message with a value of zero will send a reply. If the server to whom the message is addressed gets the message, it will send a reply regardless of the value in the <code>max-forwards</code> header field. Using a sequence of these requests, a client can identify all proxy servers involved in forwarding a certain request.
PUT	used to transmit files to a server. This method is similar to the <code>PUT</code> command used in FTP. This imposes a security threat and is therefore almost never used.
DELETE	This method asks the server to delete the file addressed in the URI. Since this method imposes a security risk no known productive HTTP servers support that method. The <code>DELETE</code> method is very similar to the <code>DELETE</code> command in FTP.
CONNECT	is a command used to create a secure socket layer (SSL) tunnel through a HTTP proxy server. A proxy server addressed with that method would open a connection to the target server and forward all data regardless of its content. That way a secure connection can be established from client to the server even though a proxy server is in use.

Server responses

As stated above, each server reply always contains a status code. Generally server replies are structured in 5 different categories. Status Codes are three digit numbers. Each category can be identified by the first digit. These Categories split up the total set of status codes by their meaning:

1xx	Informational — For example 100 Continue
2xx	Successful — For example 200 OK
3xx	Redirection — Redirects to a different URL
4xx	Client Error — For example 404 Not found or 403 Forbidden
5xx	Server Error — For example 500 Internal Server Error

Virtual Hosts

Virtual Hosts is a concept which allows multiple logical web servers to reside on one physical server, preferably with one IP Address. The different concepts are:

- A server is assigned multiple IP addresses, and each IP address is used by one single logical web server.
- A server is assigned one IP address and the different logical web servers listen to different ports. This results in URLs looking like `http://www.xyz.com:81/`
- A server is assigned one IP address. Multiple Domain Names are mapped to that IP address. All logical web servers listen to one single port. The server distinguishes requests using the `Host` field, which is mandatory for HTTP requests since HTTP/1.1.

HTTP/1.0 did not explicitly support virtual hosts. A web server managing multiple domains had to distinguish requests by the destination IP address or the port. As different ports were rarely used for virtual hosts, the server needed one IP address for each domain hosted. When the Internet began to grow rapidly, the amount of IP addresses available soon was too limited. A solution based on different ports was inconvenient and could cause confusion when a user forgot to supply the port number and received no or a wrong document.

HTTP/1.1 introduced the `Host` header field, which is mandatory in any HTTP/1.1 request. Therefore a server can now host multiple domains on the same IP address and port, by distinguishing the target of the request using the information supplied in the `Host` header field.

Content Negotiation

Usually the body of an HTTP response includes data for user interpretation. Different users might be better served with different versions of the same document. Apache can keep multiple versions of the same document, in either a different language or a different format. The included standard page displayed right after Apache is installed is an example as there are multiple versions each in a different language. Two ways can be distinguished for determining the best version for a user: server driven and client driven content negotiation.

Server Driven Content Negotiation With server driven content negotiation, the server decides which version of the requested content is sent to the client. Using the `Accept` header field, the client can supply a list of formats that would be acceptable to the user, regarding format as well as language. The server will then try to select the best suitable content.



Client Driven Content Negotiation Using server driven content negotiation, the client has no influence on the choice made by the server if none of the accepted formats of the source are available. Since it is not practicable to list all possible formats in the desired order, the client can use the `Accept` Header with the value `Negotiate`. The server will then reply with a list of available formats instead of the document. In a subsequent request the client can then directly request the chosen version.

Persistent Connections

HTTP/1.0 limited one TCP connection to last for one single request. When HTTP was developed, HTML documents usually consisted of the HTML file only, so the protocol was appropriate. As web pages grew to multimedia sites, one single page consisted of more than one document due to images, sounds, animations and so forth. A popular news web-site's index page today needs 150 different file requests to be displayed completely. Opening and closing a TCP connection for every file imposed a delay for users and a performance overhead for servers. Client and server architects soon added the header field `Connection: keep-alive` to reuse TCP connections, despite the fact that it was not part of the HTTP standard.

HTTP/1.1 therefore officially introduced persistent connections and the `Connection` header field. By default a connection is now persistent unless specified otherwise. Once either partner does not wish to use the connection any longer, it will set the header field `Connection: close` to indicate the connection will be closed once the current request has been finished. Apache offers configuration directives to limit the amount of requests for one connection and a time-out value, after which any connection has to be closed when no further request is received.

Proxy and Cache

Statistically, it is a well-known fact that a very high percentage of the HTTP traffic is accumulated by a very low percentage of the available documents on the Internet. Also a lot of these documents do not change over a period of time. Caching is technique used to temporarily save copies of the requested documents either by the client applications and/or by proxy servers in between the client application and the web server.

Proxy Servers A proxy server is a host acting as a relay agent for an HTTP request. A client configured to use a proxy server will never request documents from a web server directly. Upon each request, it will open a connection to the configured proxy server and ask the proxy server to retrieve the document on its behalf and to forward it afterwards. Proxy Servers are not limited to one instance per request. Therefore a proxy server can be configured to use another proxy server. The technique of using multiple proxy servers in combination is called cascading. Proxy Servers are used for two reasons:

1. Clients may not be able to connect to the web server directly. Often proxy servers act as intermediate nodes between private networks and public networks for security reasons. A client on the private network unable to reach the public network can then ask the proxy server to relay requests to the public network on its behalf. HTTP connectivity is then ensured.

2. Caching proxy servers are often used for performance and bandwidth saving reasons. A document often requested by multiple nodes only needs to be requested once from the origin server. The proxy server which was involved in transmitting the document can keep a local copy for a certain time to answer subsequent requests for the same document without the need to contact the source's web server. Bandwidth is saved, and performance improves as well if a higher quality connection is used between the proxy server and the clients.

Cache Control Even though caching is a favorable technique, it has its problems. When caching a document, a cache needs to determine how long that document will be valid for subsequent requests. Some information accessible to a cache is also of private or high security nature and should in no case be cached at all. Therefore cache control is a complex function that is supported by HTTP with a variety of header fields. The most important are:

If-Modified-Since A client can request documents based on a condition. If the client or proxy has a copy of the document, it can ask the server to send the document only if it has been modified since the given date. The server will reply with a normal response if the requested document has been modified, or will return "304 Unmodified" if it has not.

Expires Using this header field, a server can equip a transmitted document with something similar to a time-to-live. A client or proxy capable of caching and evaluating that header field will only need to re-request the document if the point in time appended to that header field has elapsed.

Last-Modified If the server cannot supply a certain expiration time, clients or proxies can implement algorithms based on the `Last-Modified` date sent with a document. HTTP/1.1 does not cover specific instructions on how to use that feature. Clients and proxies can be configured to use that header field as found appropriate.

Additionally, HTTP/1.1 includes several header fields for advanced cache control. Servers as well as clients are able to ask for a document not to be cached. Servers can also explicitly allow caching, and clients are even able to request a document from a caching proxy server only if it has a cached copy of the document. Clients can also inform caching proxy servers that they are willing to accept expired copies of the requested document.

2.4 Access Control and Security

2.4.1 Authorization

Although closely related, it is important to differentiate between authorization and authentication. Authentication is the process of verifying if someone is the one he pretends to be, while authorization means checking if an identified person or machine is permitted to access a resource. An HTTP server first checks if access to a resource is restricted. If the restriction



applies to persons, it requests authentication data from the client to check the user's identity. After that it checks if the authorization rules grant access to the resource for this user.

Access to a resource can be restricted to the domain or network address of the browser machine or to a particular person or group of persons. Apache determines this by parsing the global and local configuration files. If permitted by the administrator, web authors can restrict access to their documents via local configuration files which are usually named `.htaccess` (hypertext access).

Authorization rules can combine machine and person access. They apply to resources and to HTTP methods (e.g. disallow POST for unknown users). For further information on administration, take a look at the configuration section 3.2.1.

2.4.2 Authentication methods

There are many ways of checking the identity of a person who controls the browser sending an HTTP request:

- **User ID & Password:** The user has to tell the server his user ID and a secret password. The server checks the combination using its user database. If it matches, the user is authenticated.
The user database can have various forms:
 - a simple file
 - a database (local or on a different server)
 - the authentication mechanism of the operating system, a database or any application managing user data
 - a user management service like LDAP², NIS³, NTLM⁴
- **Digital signature:** The user provides a certificate pointing out his or her identity. There must be mechanisms to make sure that only the user and no one else can provide the certificate (public/private keys, key cards, etc.)

How does the server get the authentication data? There are two ways:

- HTTP authentication
- HTML Forms, Java applets and server-side scripting

Get Authentication data with HTTP

When a request applies to a protected resource, Apache sends a 401 unauthorized response to the client. This is done to inform the client that he has to supply authentication

²Lightweight Directory Access Protocol: A protocol, introduced by Netscape, to access directory servers providing user information.

³Network Information Service (formerly YP – Yellow Pages): A network service providing user information and workstation configuration data for unix systems.

⁴NT LanManager: A protocol for user authentication in SMB networks used by Microsoft Windows.

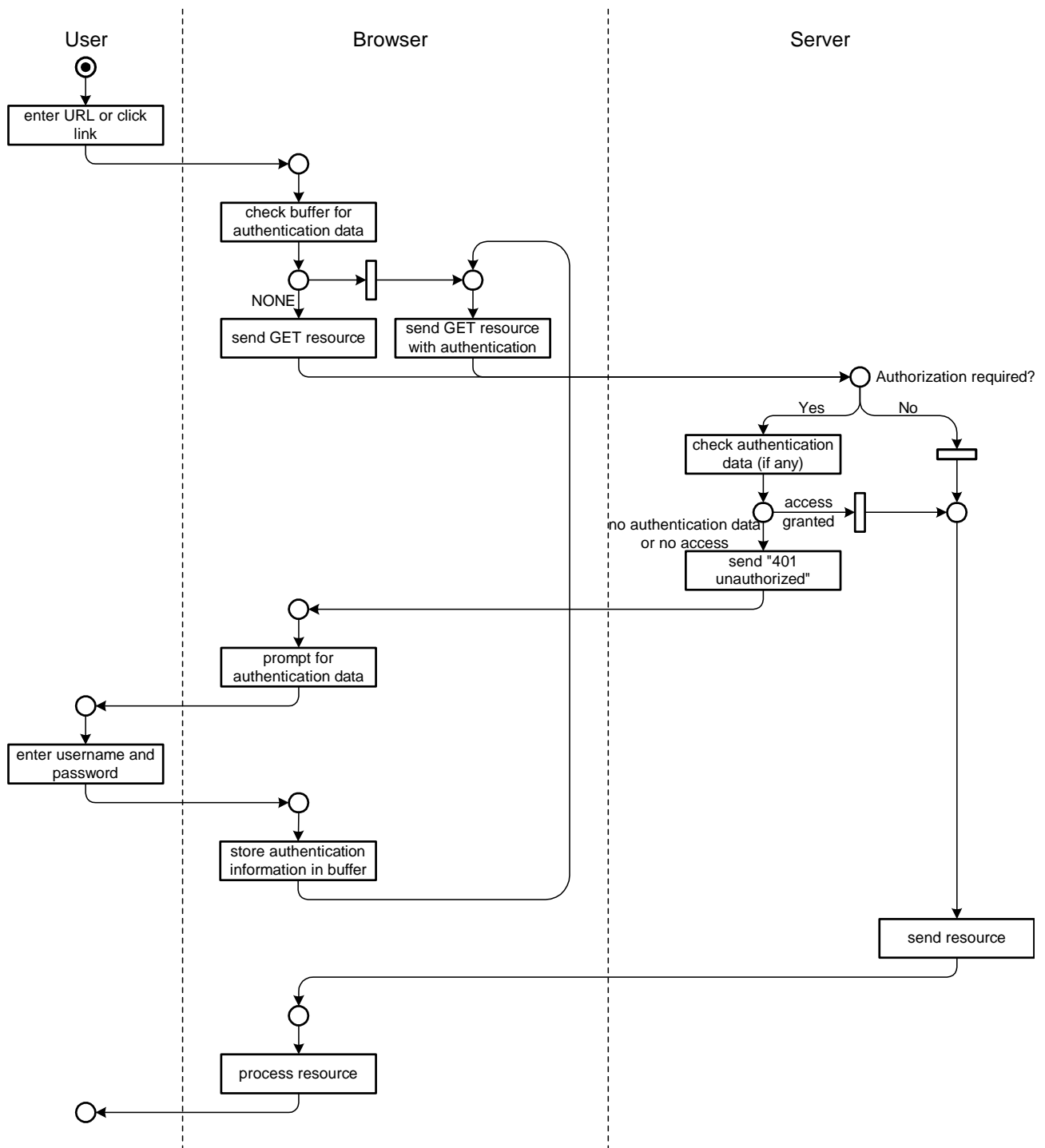


Figure 2.8: Basic Authentication process



information within his requests. Along with the 401 response the server sends the *realm*, the corresponding protected area of the website.

A browser then prompts the user for authentication data (username and password). Afterwards, the client repeats the request including the new authentication data. This data is transmitted in HTTP header fields.

Since HTTP is a stateless protocol, authentication data must be retransmitted in every request. The browser normally keeps the authentication information for the duration of the browser session. For every request concerning the same realm, the browser sends the stored authentication information (see Figure 2.8).

Generally, HTTPS should always be used for transmitting authentication data.

Basic Authentication Basic authentication is the simplest method of authentication. Although the password may be stored encrypted on the server, the browser transmits user name and password without encryption. This means, anyone listening can get the username and password, especially since they are re-sent with every request. In addition to this, the requested content is sent without encryption as well.

The solution to the problem is to use basic authentication together with encryption on the TCP layer, namely SSL. Together with HTTP, this is called HTTPS and assures that any information sent over the network is encrypted. See below in section 2.4.3.

Digest Authentication Digest authentication provides an alternative way of authentication, addressing the problem of transmitting authentication data in clear text.

The browser sends a hash value of the password and other information, an MD5 Digest, to the server. However, this method is still vulnerable to a man-in-the-middle attack.

Concluding, digest authentication is more secure than basic authentication, but not completely secure. Therefore it is rarely used. The consequence is the usage of HTTPS. For additional information on authentication look at <http://httpd.apache.org/docs/howto/auth.html>.

Get Authentication data with HTML Forms, Java applets or Java Script

Browsers usually support basic authentication. Other methods require additional software at the server side. There are several other possibilities to handle authentication data:

There is, for example, the possibility to enter authentication information into HTML forms. This information is afterwards passed to the server with HTTP POST, in the body of the request. This is different to basic authentication where the authentication data is sent in the header of the request. A CGI program can then perform the authentication check.

Another possibility is the usage of Java applets or JavaScript code. The server sends a Java applet or JavaScript code to the client. The browser executes the code which asks for authentication data and sends it to the server in an arbitrary way.

Both methods need additional software at the HTTP server to do authentication. This could for example be a CGI script that handles the content of a POST request. Additionally, as before, HTTPS is needed for a secure transmission of the data.

2.4.3 HTTPS and SSL

Security summarizes the problem of people eavesdropping the information sent. As credit card information and passwords have to be sent over the Internet, securing and therefore usually encrypting traffic becomes more important.

Authentication summarizes techniques that make sure the communication partner is who he pretends to be. It therefore helps to prevent man-in-the-middle attacks. A shopping portal or bank web site should be able to clearly identify itself, before it asks for any private information.

Securing connections

For securing connections either symmetric or public/private key algorithms can be used. With symmetric key algorithms both communication partners need the same key to encrypt and decrypt the data. That imposes a problem, as the key has to be transmitted from one partner to the other before secure communication can begin. Secure transfer of the symmetric key has to be accomplished using other security mechanisms.

Public/private key mechanisms are based on two different keys. One communication partner publishes the public key to anyone wishing to communicate. The public key is used to encrypt messages that only the owner of the private key can decrypt. Employing that technique on both sides can secure a two-way connection.

Symmetric key mechanisms usually require a smaller processing overhead than public key mechanisms. Therefore public key securing mechanisms are often used to securely transmit a symmetric key with a short validity. That secures the whole data transmission and minimizes processing overhead.

Authentication by certificates

Authenticity is more complex to accomplish. A certificate authority issues a certificate which can be checked for its validity using the certificate of the Certificate Authority. After being requested to authenticate, the communication partner will supply a certificate which can be verified using the certificate authority. Therefore the communication partner can be sure of the identity of the entity providing the certificate. However the communication partner still needs to be sure of the identity of the Certification Authority. Therefore a few companies are regarded as generally trusted authorities and any client has a list of those trusted authorities. Additional authorities that are not generally trusted have to supply a certificate for themselves that was issued by another authority that is either generally trusted or again has a certificate by a trusted authority. A new authority will therefore only need a certificate from a trusted authority to be able to issue certificates itself. A client might have to check a chain of certificates until it reaches a trusted authority that can issue a certificate for itself.

To make sure the certificate is not stolen, public/private key mechanisms are used together with the certificate. The certificate owner will provide an encrypted text using a private key that only the certificate owner is supposed to have. The certificate authority keeps the certificate including the public key but does not have knowledge about the private key. However the supplied text can be decrypted using the public key of the certificate. Therefore if the partner claiming to own the certificate can encrypt that text he is the owner of the



private key and therefore must be the entity he claims to be. Also when authenticating web servers any certificate includes a domain name. A server can only be authenticated if it can be reached via the domain name specified in the certificate.

HTTP and SSL

SSL is a protocol which can be used together with any reliable data transfer protocol like TCP. It employs mechanisms that provide Security and Authentication. SSL employs public key mechanisms to exchange symmetric session keys which are then used to encrypt any transmitted data. Certificates are used to ensure authentication. The combination of HTTP and SSL is generally referred to as secured HTTP or HTTPS. By default it uses port 443 instead of the standard HTTP port 80. Browsers will indicate to the user that a secure connection is in use and will also notify the user in case a certificate has expired or any other situation occurred that made the establishment of a secure connection impossible.

HTTPS uses certificates to publish and verify the public key that is used to exchange the symmetric key. Therefore in an HTTPS handshake, first the client requests the server's certificate, checks that against the Certificate Authority's certificate and uses the contained public key to verify the server's identity. The key is also used to exchange a "pre master secret" which is used to create the symmetric key, also referred to as the "master secret". At the end of the handshake each communication partner informs the other that future transfer will be encrypted and starts the HTTP session.

SSL-secured HTTP connections can be reused within a certain period of time to reduce the overhead that the identification check and the tunnel creation imposes.

SSL and the Internet

Today SSL is used in almost any Web Application that involves payment or private information. Almost any eCommerce website and any online banking application that is based on HTTP uses SSL to secure the connection.

SSL can be used to secure any protocol using TCP. The SSL handshake requires additional actions from client and server, so establishing an SSL connection can not be transparent to the application. A solution is to replicate server ports by tunneling the TCP connection through a SSL connection.

Even though SSL also supports client authentication which allows the web server to clearly identify the client as a person, it is rarely used, as a client certificate is required. Client certificates require the person wanting to be clearly identified to buy a certificate at a certification authority. As hardly any company wants to force their customers to spend money just to be able to buy their products, usually username and password based methods are used to identify the client. However, the transmission of the user name and password is usually secured using SSL.

2.5 Session Management

2.5.1 HTTP — a stateless protocol

HTTP is a stateless protocol. That means each HTTP request can be treated as a "new" request with no relations to previous requests. The servers doesn't have to keep any information about previous requests from the client or any session information at all. The advantage of this design is that no session information has to be kept at the server. This allows simple and fast servers.

To implement client-server applications that are more complex and therefore need session information, the client has to send at least a session identification to the server with every request.

2.5.2 Keep the state

If a server application, for example a CGI program, needs to keep and manage state information, for example a session, it has to get the complete state information with every request or map an incoming request to a session. So the client has to transmit all state data or a state ID with every request.

There are several possibilities to achieve that :

HTML Forms

A browser sends all data of a HTML form with a POST or GET request. The server sends a form including state data stored in hidden form fields. Hidden form fields are not visible in the browser, but can be seen in the source HTML code. Normally this information will not be altered and just sent back to the server with every POST or GET of the form. A problem lies within the fact that an experienced user can alter the information.

HTML links

It is also possible to generate links in HTML documents containing a state ID. This could simply be attached to the request URL on every GET request. In contrast to the HTML forms, this works with every kind of link in an HTML page.

Cookies

Another possibility to store state information is a concept called cookies, invented by Netscape in 1995. Cookies are a possibility for the server to store state information on the client side.

The server stores small text fields (cookies) at the browser by sending them in the header of a response. Afterwards, the browser sends them in the HTTP header of every request to the server. A cookie has additional information like a validity period, domain information describing the scope of the cookie on the website, etc. The browser processes this information



to decide which cookies to send with which request. If the client gets a newer cookie with the same name from the same server, the old one is replaced.

Cookies can be used to collect user data and to create detailed profiles of the user. For example using the information gathered using cookies one can keep track on where a user surfs in the Internet and in what he is interested in. That is the reason why cookies are often criticized.

"The state object is called a cookie, for no compelling reason." (*Netscape Cookie Specification*)

Java applet / JavaScript

The server can send a Java applet or a JavaScript program to the browser which executes this program. Such a program can then store the state information and communicate via POST or GET or even with an arbitrary protocol with a program at the server machine.

However, the client browser must support execution of java or script code. Many users don't like to execute java applets or script code on their machine, since they could be dangerous if the source cannot be trusted.

2.6 Dynamic Content

A simple web server delivers static documents which are stored on the server as files. The author has to change or update these documents manually. As the Internet and also the demand for high-level multimedia content grew, the need for dynamic web pages arose. Web based applications ranging from a personal address book to online banking applications or a big portal that allows personalization can either be achieved by altering the server's functionality, via server-side or client-side scripting. As this document is focused on the Apache Web Server, client-side scripting like Java Script will not be covered.

Basically, web clients are designed to display pages that they received as reply to a request. The server can therefore either send a static file or generate a page based on data from external sources like a database and/or on data received by the client. The client sends data of HTML Forms with GET or POST requests or via proprietary ways implemented in Java Applets.

The first technology that was available to offer dynamic web content was CGI (Common Gateway Interface). CGI enables the web server to execute external programs, which output HTML code.

Alternatively, extending the functionality of Apache to implement dynamic Web Applications would include the need to develop additional modules, which is covered later in this document. Writing a separate module for each web application allows for good performance, but can be rather complicated and expensive. As each new module would either need a server restart or even a complete recompilation of the server source code, that solution is rather inconvenient, especially in ISP environments where multiple parties share one server.

Therefore most web applications are usually implemented using a scripting language which will be executed by an interpreter called via CGI or by an interpreter module.

2.6.1 Server-side Scripting

An easy way to enable a server to deliver dynamic content is to employ server-side scripting. As mentioned above one of the first technologies to enable the server to provide dynamic content was CGI. To enable scripting using CGI, the web server executes an external program, which is able to interpret scripts that return HTML code to the server which then forwards the output to the client.

Optionally the server is enhanced using modules to support script languages that are interpreted within the server's context when a request is handled. The module supplies an execution environment for the script and offers API functions to enable the script to access external data sources and data received from the client.

In Apache, each script-language-enabling module usually registers a separate MIME-type and defines a file extension for files supposed to be executed by the module's content handler.

Generally Server-side Scripting can be subdivided in scripts embedded in HTML files and scripts that output complete HTML documents.

HTML files with embedded scripts

In this case the script is embedded in the HTML document file on the web server. Basically the document includes HTML code like a static page. Within certain tags that are recognized by the script-executing module, the developer can insert scripting instructions that will output HTML code replacing the scripting instructions. The output these instructions generate can be based on external data sources such as a database or on input received by the client with the request.

Server-Side Includes (SSI) One example for scripting embedded in HTML code is "Server Side Includes" also referred to as SSI. SSI enables basic commands like assigning values to variables, accessing own and system's variables, doing basic computation and even execute system commands like on a command line and printing their output to the web page.

One common use for SSI is to include another file into a document. Therefore SSI can be used to decorate CGI pages with a header and footer and therefore save some work when programming CGI pages by excluding static HTML output from the CGI script. (See below for information on CGI).

SSI commands can be included in HTML pages using special commands within comment tags, which a browser will ignore in case the server is accidentally unable to interpret the script:

```
<!--#your_command_here -->
```

Other examples for HTML enhancing script languages are JSP, PHP or ASP.



Programs generating complete HTML documents

For complex web applications HTML enriching script languages tend to be not performant enough. Therefore compiled programs or scripts are used that output complete HTML documents. Files containing scripting commands may not include static HTML, which relieves the server from having to parse the whole document for scripting instructions. Another reason for these script languages to be a lot faster than languages allowing simple HTML code in their files is that some of them can be compiled and therefore run a lot faster than scripts that have to be interpreted.

Examples for that category are CGI programs and Java Servlets. Certain flavours of CGI and Java Servlets have to be compiled and therefore gain performance.

Usually such technologies are a lot more powerful and therefore allow for more flexibility. Communication with external programs is easier and some even start processes that run in the background and are responsible for keeping state information. CGI is used to start external programs (like directory info, e.g. `ls -l` or `dir`) and send their output to the client. The drawback of CGI's approach is the fact that the program is started in a separate context (e.g. a process) when a request is received which can drastically affect performance. Additionally communication is restricted to the use of environment variables and the use of STDIN and STDOUT.

Scripting technologies employing modules to interpret scripts overcome these limitations as scripts are interpreted in the server context. Therefore no context switch is needed and communication between server and script is only limited by the server's and the module's API. With that technology, a request arriving at the web server will trigger the execution or interpretation of the script and forward all output to the client. Therefore the script outputs HTML code, which is sent to the client and interpreted by the client's browser as if it was a static HTML page.

The complexity of a script application is usually not limited by the script language capability but by the servers performance. For very complex applications it might be worth implementing an add-on module compiled into the web server and therefore usually gains performance as no interpretation nor a context switch is needed. Additionally the possibility to use the complete server API without the restrictions a interpreting module may imply allows for more powerful applications.

Chapter 3

The Apache HTTP Server

3.1 Overview

3.1.1 History

The Beginning

Apache is an offspring of the NCSA httpd web server designed and implemented by Rob McCool. He made the NCSA server as the web server market was dominated by complex and heavyweight server systems. Many people wanted a simple and small server that would be adequate for the needs of a simple web site. However Rob McCool was not able to continue his work on the NCSA server due to work commitments. He left the project and abandoned the server. At that stage a lot of people were already using the NCSA server. As with any software system, a lot of people patched the server to suit their own demands and to get rid of bugs. In 1995 Brian Behlendorf started to collect those patches and extensions and founded a mailing list that was solely used for that purpose of exchanging. A group of 8 people that formed that mailing list community released the very first version of Apache. Due to its nature of consisting of patches and extensions to the NCSA server, the name Apache is actually a derivate of "a patchy server".

Evolution

The first version that was released by the mailing list was version 0.6.2. One member of the growing group of developers, Robert Thau, designed a new server architecture that was introduced with version 0.8.8. On December 1st in 1995, Apache version 1.0 was released and only took a year to become more popular and more widely used than the older and formerly popular NCSA server.

During the next years the group began to grow and Apache received many new features and was ported to different operating systems.

In 1999 the group founded the Apache Software Foundation to form a non-profit company. In March 2000 the ApacheCon, a conference for Apache developers, was held for the first time.



Apache 2

On the ApacheCon conference in March 2000, Apache version 2.0 Alpha 1 was introduced to the public. The version 2 of Apache again introduced a complete redesign of the server architecture. Apache 2.0 is easier to port to different operating systems and is considered so modular that it does not even have to be used as a web server. By designing appropriate modules the Apache 2.0 core could be used to implement any imaginable network server.

Today both versions of Apache, version 1.3 and version 2.0 exist. Even though people are encouraged to use the newer version many still use version 1.3 which is still being further developed and supported.

For more information on Apache history see the Apache History Project's website at <http://www.apache.org/history/>.

3.1.2 Features

Both versions of Apache used today form the biggest share of the web server market. Even though the fact that Apache is free is an influencing fact, the main reason for Apache's success is its broad range of functionality.

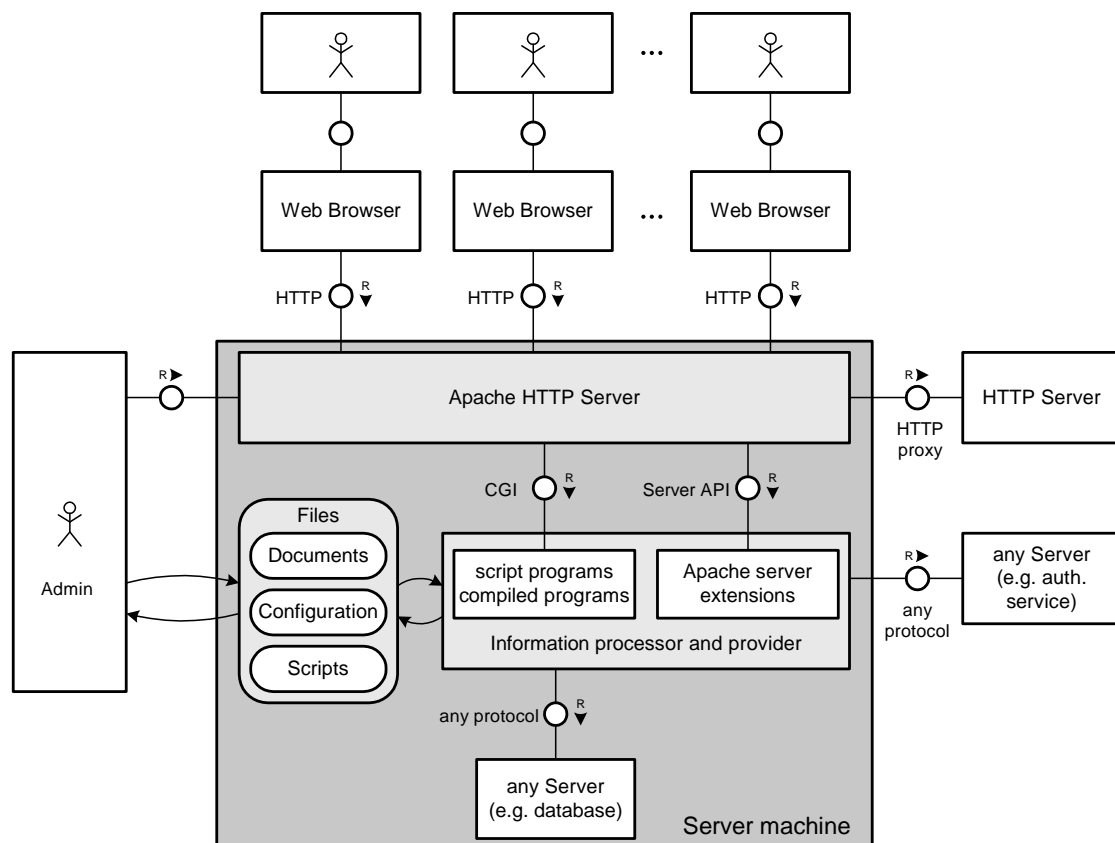


Figure 3.1: The Apache HTTP Server in its environment

Apache is a server that supports concurrency and can therefore serve a big number of clients. The number of clients that can be served concurrently is limited only by the underlying

hardware and operating system. The server can be easily configured by editing text files or using one of the many GUIs that are available to manage these. The server can be re-configured without having to stop the server. Due to its modularity, many features that are necessary within special application domains can be implemented as add-on modules and plugged into the server. To support that, a well documented API is available for module developers. Its modularity and the existence of many free add-on modules makes it easy to build a powerful web server without having to extend the server code. Using many of the available server based scripting languages, web based applications can be developed easily. When using scripting languages or add-on modules Apache can even work with other server applications like databases or application servers. Therefore Apache can be used in common multi-tier scenarios. Additionally Apache is completely HTTP 1.1 compliant in both of the current versions. The installation is easy and ports for many popular platforms are available.

The block diagram in figure 3.1 shows an Apache HTTP Server in its environment. In comparison to the simple HTTP Server system shown in figure 2.1, we see a administrator, configuration by files and server extensions using CGI or or the Server API. These extensions (Information processor and provider) can access any resources at the server machine or via network at any remote machine.

3.2 Using Apache

3.2.1 Configuration

As the usage and administration of Apache is covered by many other publications, this document will only give an overview, as it is necessary to understand how Apache works as a basis for the following parts of this document.

There are basically four ways to configure Apache:

1. Building / Installing Apache

Configuring Apache by choosing modules, setting the compiler and flags used for building, selecting the installation path and so on. Further information can be found in [1].

2. Command-line parameters

Configuring Apache at start-up. The available command-line options can also be found in [1].

3. Global configuration files

Apache uses a global configuration file, which is processed at the server's start-up. It is named `httpd.conf`, by default, and resides in the `conf/` directory within the server root directory.

4. Local configuration files

Apache can also be configured by local configuration files, named `.htaccess` by default, on a per-directory basis. These files are taken into account while processing a request when Apache walks through the file system hierarchy to find the requested documents. For example they enable different authors to configure their web-space on their own.

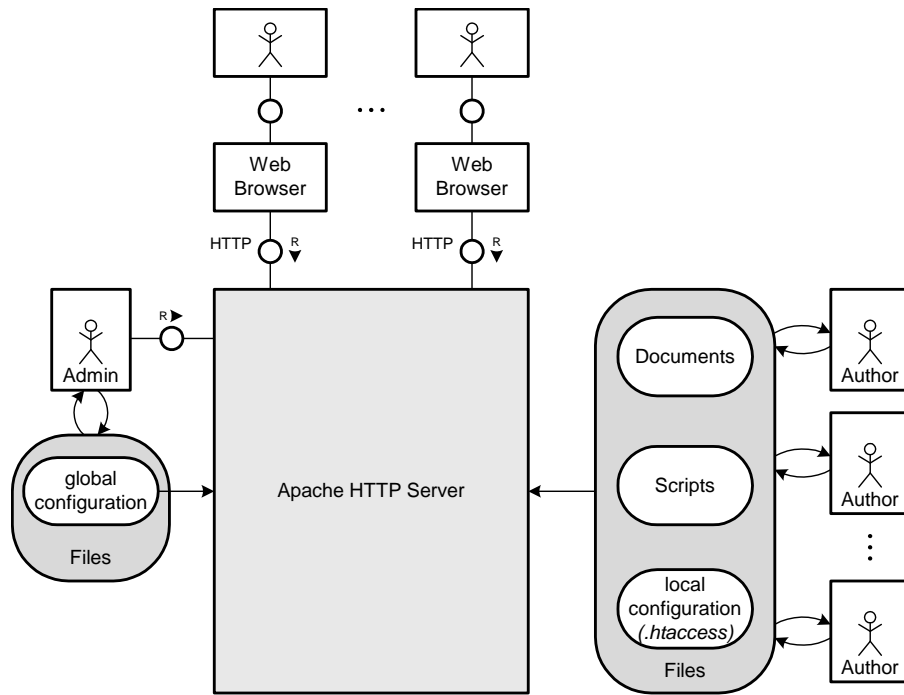


Figure 3.2: Configuring the Apache HTTP Server via configuration files

The last two possibilities describe the configuration of Apache via text files as it can be seen in figure 3.2. The following parts will focus on explaining the structure of these configuration files and give examples on how to use them.

Global and local configuration files

Global configuration The configuration directives in the main Apache server configuration file `httpd.conf` are grouped into three basic sections:

1. Directives that control the operation of the Apache server process as a whole (the 'global environment').
2. Directives that define the parameters of the 'main' or 'default' server, which responds to requests that aren't handled by a virtual host. These directives also provide default values for the settings of all virtual hosts.
3. Settings for virtual hosts, which allow HTTP requests to be sent to different IP addresses and/or hostnames and have them handled by the same Apache server instance.

Directives placed in the main configuration files apply to the entire server. If you wish to change the configuration for only a part of the server, you can scope your directives by placing them in `<Directory>`, `<DirectoryMatch>`, `<Files>`, `<FilesMatch>`, `<Location>`, and `<LocationMatch>` sections. These sections limit the application of the directives which they enclose to particular file system locations or URLs.

The `<Directory>` sections apply to 'real' directories at any position in the file system, whereas `<Location>` sections apply to the Request URIs.

Apache has the capability to serve many different websites with different host names simultaneously. This is called Virtual Hosting. Therefore directives can also be scoped by placing them inside `<VirtualHost>` sections, so that they will only apply to requests for a particular website.

In the global server configuration file the webmaster can configure the server with the provided directives and limit the options the users have in the per-directory configuration files (`.htaccess` files).

Changes to the main configuration files are only recognized by Apache when it is started or restarted.

Local configuration Apache allows for decentralized management of configuration via special files placed inside the web tree, allowing users to configure their web space on their own to a certain degree, that is granted by the administrator in the main configuration files. The special files are usually called `.htaccess`, but any name can be specified in the `'AccessFileName'` directive. Since `.htaccess` files are read on every request, changes made in these files take immediate effect. Directives placed in `.htaccess` files apply to the directory where you place the file and all sub-directories, overwriting all directives in previous files.

Because `.htaccess` files are evaluated for each request, you don't need to restart the Apache server when you make a change. This makes them useful if multiple users are sharing a single Apache system. Another benefit of these files is, a syntax error in `.htaccess` files affects only a part of the server, but the system can continue working.

The two main disadvantages of using `.htaccess` are the performance impact and the extending of access to the configuration data to others. The first is somewhat manageable through the judicious use of the `AllowOverride` directive, the latter is a matter of trust and performing risk assessment.

Syntax

Apache gets its instructions through configuration directives used in the configuration files. There are two types of directives, simple ones and sectioning directives, which again can contain one or more directives.

Apache processes the files on a line by line reading any line that is neither empty nor a comment line beginning with the character `'#'`. The first word in such a line is the name of the directive whereas the remaining ones are treated as the parameters of the directive. To use more than one line for the parameters of a directive, the backslash `'\'` may be used as the last character on a line to indicate that the parameters continue on the next line.

Apache distinguishes between several contexts in which a directive can be used. Each directive is only allowed within a fixed set of contexts.

Global Configuration The 'Per-Server Context' applies to the global `httpd.conf` file (the file name can be overridden by the `-f` option on the `httpd` command line) and is divided into five sub-contexts:



1. The global context which contains directives that are applied to the default or main server.
2. (`<VirtualHost>`) The virtual host sections contain directives that are applied to a particular virtual server.
3. (`<Directory>`, `<DirectoryMatch>`) The directory sections contain directives that are applied to a particular directory and its subdirectories.
4. (`<Files>`, `<FilesMatch>`) The file sections contain directives that are applied to particular files.
5. (`<Location>`, `<LocationMatch>`) The URL sections contain directives that are applied to a particular URL and its sub-areas.

Directives placed in the main configuration file apply to the entire server. To change the configuration for only a part of the server, place your directives in the appropriate context. Some section types can also be nested, allowing for very fine grained configuration. Generally, all directives can appear in the global configuration file.

Local Configuration The 'Per-Directory Context' applies to the local `.htaccess` files, which only allow configuration changes using directives of the following five sub-contexts:

1. (`AuthConfig`) The Authentication context contains directives that control authorization.
2. (`Limits`) The Limit context contains directives that control access restrictions.
3. (`Options`) The Option context contains directives that control specific directory features.
4. (`FileInfo`) The File information context contains directives that control document attributes.
5. (`Indexes`) The Index context contains directives that control directory indexing.

Directives placed in `.htaccess` files apply to the directory where you place the file, and all sub-directories. The `.htaccess` files follow the same syntax as the main configuration files. The server administrator further controls what directives may be placed in `.htaccess` files by configuring the `'AllowOverride'` directive in the main configuration files.

Further information and a list of directives with allowed contexts can be found in [1] and at <http://httpd.apache.org/docs/>.

How Apache determines the configuration for a request

1. Determine virtual host
The corresponding virtual host to the URI has to be found.

2. Location walk

The configuration for the URI has to be retrieved before the URI is translated.

3. Translate Request URI (e.g.: `mod_rewrite`)

The modules have the opportunity to translate the Request URI into an actual filename.

4. Directory walk beginning from root (/) directory, applying `.htaccess` files

Apache reads the configuration of every section of the path and merges them.

5. File walk

Apache gets the configuration for the files.

6. Location walk, in case Request URI has been changed

When Apache determines that a requested resource actually represents a file on the disk, it starts a process called 'directory walk'. Therefore Apache has to check through its internal list of `<Directory>` containers, built from the global configuration files, to find those that apply. According to the settings in the global configuration files Apache possibly searches the directories on the file system for `.htaccess` files.

Whenever the directory walk finds a new set of directives that apply to the request, they are merged with the settings already accumulated. The resulting collection of settings applies to the final document, assembled from all of its ancestor directories and the server's configuration files.

When searching for `.htaccess` files, Apache starts at the top of the file system. It then walks down the directories to the one containing the document. It processes and merges any `.htaccess` files it finds that the global configuration files say should be processed.

The sections are merged in the following order:

1. `<Directory>` (except regular expressions) and `.htaccess` are merged simultaneously with `.htaccess` overriding `<Directory>` sections, if allowed
2. `<DirectoryMatch>` and `<Directory>` with regular expressions
3. `<Files>` and `<FilesMatch>` merged simultaneously
4. `<Location>` and `<LocationMatch>` merged simultaneously

Each group is processed in the order that they appear in the configuration files. Only `<Directory>` is processed in the order "shortest directory component to longest". If multiple `<Directory>` sections apply to the same directory they are processed in the configuration file order. The configuration files are read in the order `httpd.conf`, `srm.conf` and `access.conf` (`srm.conf` and `access.conf` are deprecated and are kept only for backward-compatibility).

Sections inside `<VirtualHost>` sections are applied after the corresponding sections outside the virtual host definition. This way, virtual hosts can override the main server configuration. Finally, later sections override earlier ones.

For details see sections 4.4.4 and 4.5.



Example Configuration

httpd.conf:

```
#####
# Section 1: Global Environment
# Many of the values are default values, so the directives could be omitted.
ServerType standalone
ServerRoot "/etc/httpd"
Listen 80
Listen 8080
Timeout 300
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
MinSpareServers 5
MaxSpareServers 10
StartServers 5
MaxClients 150
MaxRequestsPerChild 0

#####
# Section 2: "Main" server configuration
ServerAdmin webmaster@foo.org
ServerName www.foo.org
DocumentRoot "/var/www/html"

# a very restrictive default for all directories
<Directory />
    Options FollowSymLinks
    AllowOverride None
</Directory>

<Directory "/var/www/html">
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>

#####
# Section 3: virtual hosts
<VirtualHost www.foo.dom:80>

# all hosts in the hpi.uni-potsdam.de domain are allowed access;
# all other hosts are denied access
<Directory />
    Order Deny,Allow
    Deny from all
    Allow from hpi.uni-potsdam.de
</Directory>

# the "Location" directive will only be processed if the module
# mod_status is part of the server
<IfModule mod_status.c>
```

```
<Location /server-status>
    SetHandler server-status
    Order Deny,Allow
    Deny from all
    Allow from .foo.com
</Location>
</IfModule>
</VirtualHost>
```

3.2.2 Performance

Means to slow down a server

(to be completed)

(too few processes, insufficient memory, too many checks (for example: check for symbolic links), complex configuration (for example: too many .htaccess files, too many modules)

Configuration

(to be completed)

(choose proper configuration for intended usage)

3.3 Extending Apache: Apache Modules

3.3.1 Introduction

Modules are pieces of code which can be used to provide or extend functionality of the Apache HTTP Server. Modules can either be statically or dynamically included with the core. For static inclusion, the module's source code has to be added to the server's source distribution and to compile the whole server. Dynamically included modules add functionality to the server by being loading as shared libraries during start-up or restart of the server. In this case the module `mod_so` provides the functionality to add modules dynamically. In a current distribution of either Apache 2.0 or Apache 1.3, all but very basic server functionality has been moved to modules.

Modules interact with the Apache server via a common interface. They register handlers for hooks in the Apache core or other modules. The Apache core calls all registered hooks when applicable, that means when triggering a hook. Modules on the other hand can interact with the server core via the Apache API. Using that API each module can access the server's data structures, for example for sending data or allocating memory.

Each module contains a module-info, which contains information about the handlers provided by the module and which configuration directives the module can process. The module info is essential for module registration by the core.

All Apache server tasks, be it master server or child server, contain the same executable code. As the executable code of an Apache task consists of the core, the static modules and the dynamically loaded ones, all tasks contain all modules.

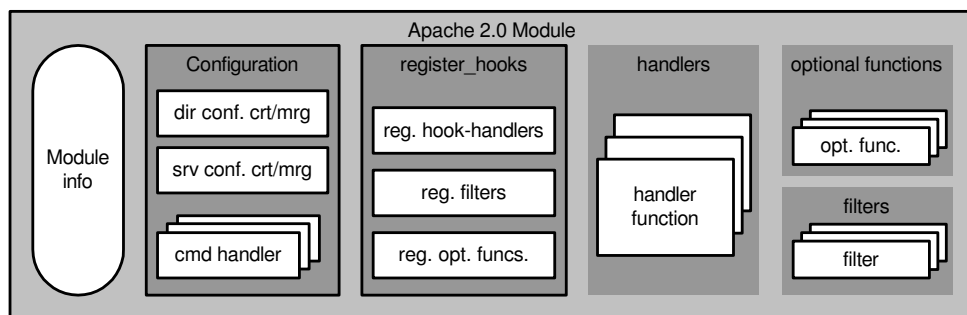


Figure 3.3: Apache 2 Module structure

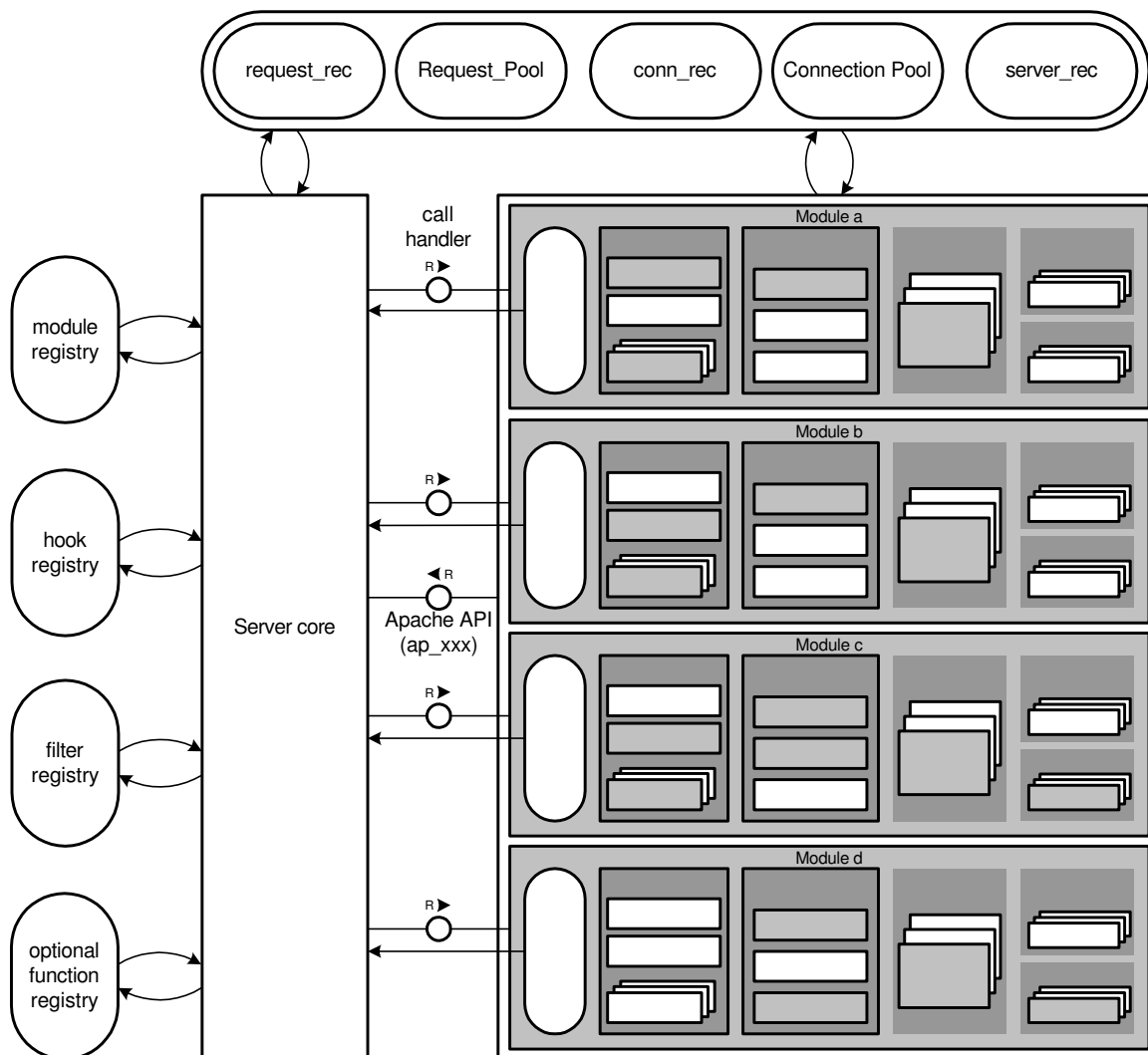


Figure 3.4: Interaction of Core and Modules

As you can see in figure 3.4, Modules and the Core can interact in two different ways. The server core calls module handlers registered in its registry. The modules on the other hand can use the Apache API for various purposes and can read and modify important data structures like the request/response record `request_rec` and allocate memory in the corresponding pools.

3.3.2 Types of Handlers

A module can provide different kinds of handlers:

- Handlers for hooks
- Handlers for dealing with configuration directives
- Filters
- Optional functions

Handlers for hooks

A hook is a transition in the execution sequence where all registered handlers will be called. It's like triggering an event which results in the execution of the event handlers. The implementation of a hook is a hook function named `ap_run_HOOKNAME` which has to be called to trigger the hook.

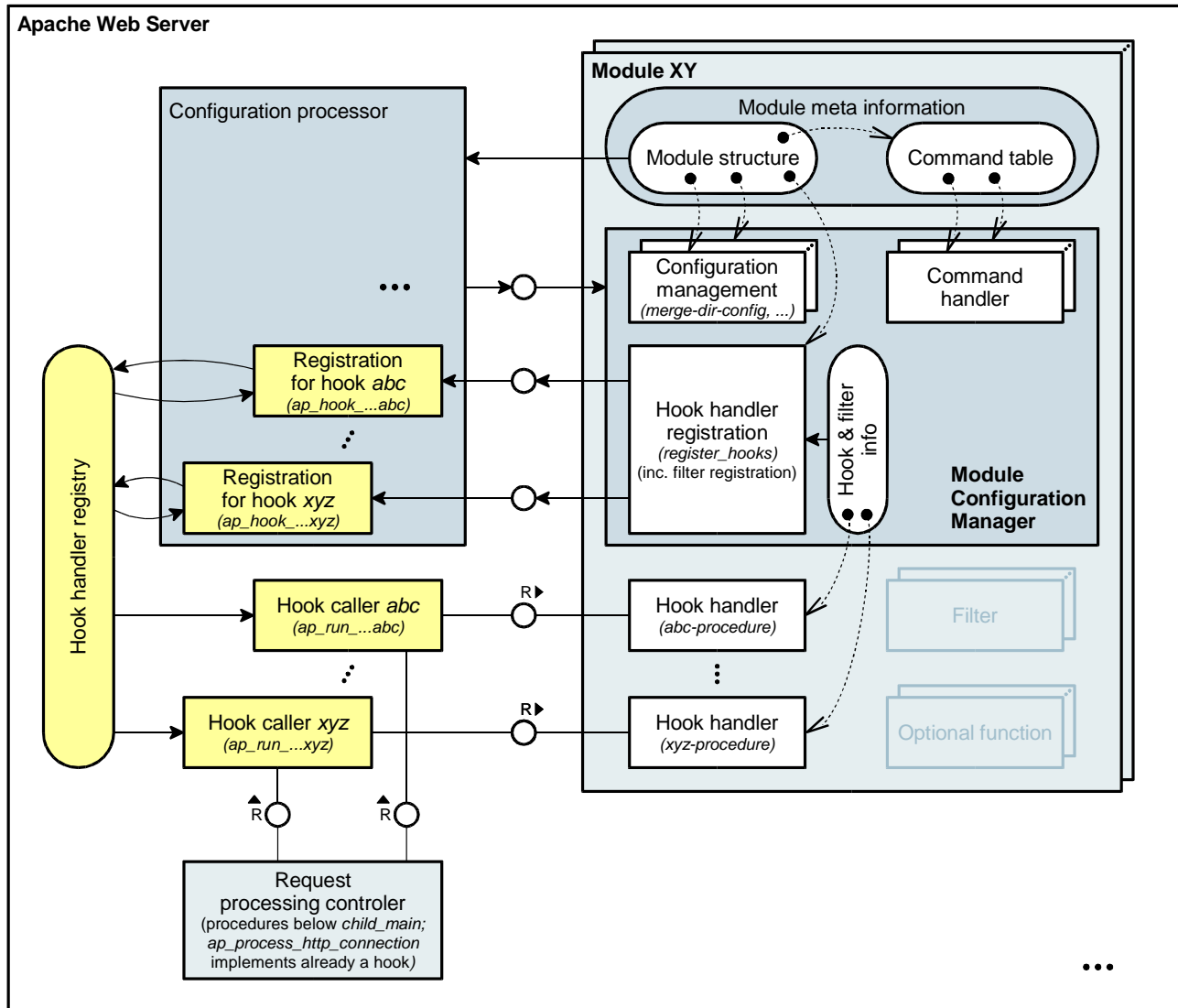
Two types of calling handlers for a hook can be distinguished:

RUN_ALL/VOID: The core calls all registered handlers in their order regardless whether they can complete the task or refuse to complete it, unless an error occurs.

RUN_FIRST: The core calls the registered handlers in their order until one module can complete the task or an error occurs.

Each module has to register its handlers for the hooks with the server core first before the server core can call them. Handler registration is different in Apache 1.3 and 2.0. Apache 1.3 provided 13 predefined hooks. Registration of the module's handlers was done automatically by the core by reading the module info while loading modules. In Apache 2.0, the module info only contains references to four handlers for predefined hooks used for configuration purposes only. All other hook handlers are registered by calling the `register_hooks` function each module has to provide. This makes it easier to provide new hooks without having to alter the Apache module interface. A module can provide new hooks for which other new modules can register hooks as well.

Figure 3.5 shows how hooks and handlers interact in Apache: A hook ABC has to be defined by some C macros (`AP_DECLARE_HOOK`, etc, see bottom line). This results in the creation of a registration procedure `ap_hook_ABC`, a hook caller procedure `ap_run_ABC` and an entry in the hook handler registry which keeps information about all registered handlers for the hook with their modules and their order. The module (meta) info at the top points to the hook handler registration procedure (`register_hooks`) which registers the handlers for the



Agent/Storage definition is generated by processing the C-MACROS
 AP_DECLARE_HOOK, APR_IMPLEMENT_HOOK_(VOID | RUN_FIRST | RUN_ALL), APR_HOOK_STRUCT & APR_HOOK_LINK

Figure 3.5: Apache 2.0 hook mechanism

hooks calling the `ap_hook_XXX` procedures. At the bottom, an agent called “request processing controller” is a representative of all agents triggering hooks by calling the `ap_run_XXX` procedures which read the hook handler registry and call all or one registered handler.

The order of calling handlers for a hook can be important. In Apache 1.3, the order of the module registration determined the order in which their handlers would be called. The order could be altered in the configuration file but was the same for all 13 hooks. In Apache 2, this has changed. The hook registry can store an individual order of handlers for each hook. By registering a handler for a hook using the `ap_hook_XXX` procedure, a module can supply demands for its position in the calling sequence. It can name modules that’s handlers have to be called first or afterwards, or it can try to get the first or the last position.

Handlers for Configuration Directives

A module can provide an own set of directives which can be used in the configuration files. The configuration processor of the core therefore delegates the interpretation of a directive to the corresponding command handler which has been registered for the directive. In figure 3.5 the module (meta) info at the top points to the configuration management handlers of the module (create-dir-config, merge-dir-config, etc.) and to the command table which contains configuration directives and the pointers to the corresponding command handlers.

The configuration management handlers have the purpose to allocate memory for configuration data read by the command handlers and to decide what to do if configuration parameters differ when hierarchically merging configuration data during request processing.

Optional functions

An Apache 2.0 module can also register optional functions and filters. Optional Functions are similar to hooks. The difference is that the core ignores any return value from an optional function. It calls all optional functions regardless of errors. So optional functions should be used for tasks that are not crucial to the request–response process at all.

3.3.3 Content Handling

The most important step in the request–response loop is calling the content handler which is responsible for sending data to the client.

In Apache 1.3, the content handler is a handler very much like any other. To determine which handler to call Apache 1.3 uses the `type_checker` handler which maps the requested resource to a mime-type or a handler. Depending on the result, the Apache Core calls the corresponding content handler which is responsible for successfully completing the response. It can write directly to the network interface and send data to the client. That makes request handling a non-complex task but has the disadvantage that usually only one module can take part in handling the request. If more than one content handler have been determined for the resource, the handler that was registered first is called. It is not possible that one handler can modify the output of another without additional changes in the source code.



Apache 2.0 extends the content handler mechanism by output filters. Although still only one content handler can be called to send the requested resource, filters can be used to manipulate data sent by the content handler. Therefore multiple modules can work cooperatively to handle one request. During the mime-type definition phase in Apache 2.0 multiple filters can be registered for one mime-type together with an order in which they are supposed to handle the data. Each mime-type can be associated with a different set of modules and a differing filter order. Since a sequenced order is defined, these filters form a chain called the output filter chain.

When the Content handler is called, Apache 2.0 initiates the output filter chain. Within that chain a filter performs actions on the data and when finished passes that data to the next filter. That way a number of modules can work together in forming the response. One example is a CGI content handler handing server side include tags down the module chain so that the include module can handle them.

3.3.4 Apache 2 Filters

Apache 2 Filters are handlers for processing data of the request and the response. They have a common interface and are interchangeable.

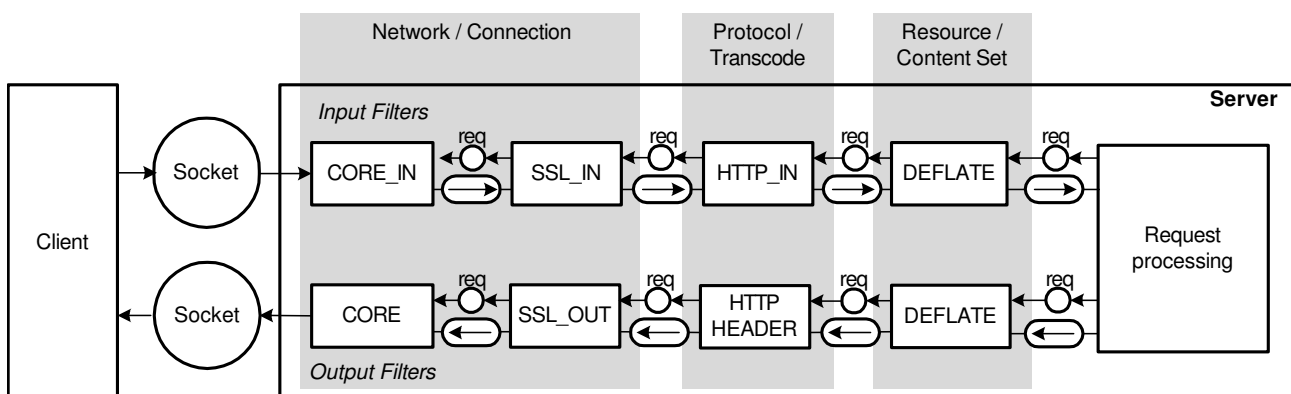


Figure 3.6: Apache Filters: The Input / Output filter chain

In figure 3.6 you see two example filter chains: The input filter chain to process the data of the request and the output filter chain to process the data of the response (provided by the content handler). The agent “Request processing” triggers the input filter chain while reading the request. An important use of the input filter chain is the SSL module providing secure HTTP (HTTPS) communication.

The output filter chain is triggered by the content handler. In our example, the Deflate output filter compresses the resource depending on its type.

To improve performance, filters work independently by splitting the data into buckets and brigades (see figure 3.7) and just handing over references to the buckets instead of writing all data to the next filter’s input (see figure 3.8). Each request or response is split up into several brigades. Each brigade consists of a number of buckets. One filter handles one bucket at a time and when finished hands the bucket on to the next filter. Still the order in which the filters hand on the data is kept intact.

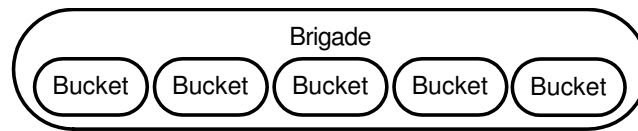


Figure 3.7: Apache Filters: A Brigade contains a series of buckets

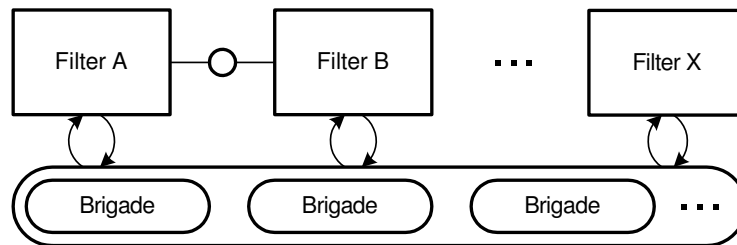


Figure 3.8: Apache Filters: A Filter chain using Brigades for data transport

Besides separating filters into input and output filters, 3 different categories can be distinguished:

1. Resource and Content Set Filters

Resource Filters alter the content that is passed through them. Server Side Includes (SSI) or PHP scripting are typical examples.

Content Set Filters alter the content as a whole, for example to compress or decompress it (Deflate).

2. Protocol and Transcode Filters

Protocol Filters are used to implement the protocol's behavior (HTTP, POP, ...). That way future versions of HTTP could be supported.

Transcode Filters alter the transport encoding of request or response. For example, the chunk output filter splits a resource into data chunks which it sends to the client one after another .

3. Connection and Network Filters

Connection Filters deal with establishing and releasing a connection. For example, establishing an HTTPS connection requires a special handshake between client and server. They may also alter the content, in the HTTPS example by encrypting and decrypting data.

Network Filters are responsible for interacting with the operating system to establish network connections and complete associated tasks. To support protocols other than TCP/IP, only a module implementing an input and output filter for the specific connection protocol is needed.

3.3.5 Predefined Hooks

Even though in Apache 2.0 handlers for hooks are registered differently from Apache 1.3, the predefined hooks are very alike in both versions and can be distinguished into 3 different categories by their purpose and their place in the runtime sequence:



1. Managing and processing configuration directives
2. Start-up, restart or shutdown of the server
3. Processing HTTP requests

1. Configuration Management Hooks

During start-up or restart, the Apache master server reads and processes the configuration files. Each modules can provide a set of configuration directives. The configuration processor of the core will call the associated command handler every time it encounters a directive belonging to a module. To prepare resources for storing configuration data, a module can register handlers for the following hooks:

Create per server config structure

If a module needs configuration data stored on a per server basis it can use this handler for that task. It is purely used to allocate and initialize memory for a per server configuration structure. It is called once for the main server and once for each virtual host.

Create per-directory config structure

A module can allocate and initialize memory it requires for per directory configuration data. This handler is called multiple times. Once after the per server configuration structure has been initialized for the main host and once for each per directory directive specific to that module.

Command handlers

In the command table each module supplies a mapping of configuration directives to functions that will handle their parameters. Also the module has to state under which circumstances each directive is permitted or should be overridden by another configuration setting.

Configuration data is organized hierarchically. Rules have to be defined in which cases a configuration parameter of a lower (more specific) level may override the parameter of a higher level. The “merge” handlers can be used for this task.

Merge per-server config structure

If during the creation of the per server configuration structure any options have been left out for a virtual host, here the module is given a chance to fill blank options with configuration data from the main server. This handler is called once for each virtual host.

Merge per-directory config structure

This handler is used to merge directory configuration data with per server configuration data as needed. If configuration data differs for the virtual host and a directory then the task of the module when called by this handler is to compute the absolute configuration data valid for that specific request. This handler is called during start-up and once for each request the server processes. Here files like .htaccess are included in the information processed. The fact that this handler is also called once per request lets website administrators change the access files during server runtime, so that they can take effect during the next request.

For more information about Apache configuration and the configuration processor, consult sections 3.2.1 and 4.5.

2. Start-up, restart or shutdown of the server

Apache is a multitasking server. During start-up and restart, there is only one task performing initialization and reading configuration. Then it starts spawning child server tasks which will do the actual HTTP request processing. Depending on the multiprocessing strategy chosen, there may be a need for another initialization phase for each child server to access resources needed for proper operation, for example connect to a database. If a child server terminates during restart or shutdown, it must be given the opportunity to release its resources.

pre_config

This hook is triggered after the server configuration has been read, but before it has been processed during start-up and during the restart loop (see also figure 4.16 on page 86). The pre_config handler is executed by the master server task and can take advantage of the privileges (if executed by root/administrator).

open_logs

A module can register a handler for this hook if it needs to open log files or start a logging process. Some MPMs (see section 4.3.4) also use this hook to access resources like server sockets.

post_config (Apache 1.3: Initializer)

This hook is triggered by the master server task after the server configuration has been read and processed during start-up and during the restart loop (see also figure 4.16 on page 86).

pre_mpm (internal)

This internal hook is triggered by the master server before starting the child servers. As it lies in the responsibility of a MPM, the core registers handlers for the hook, for example to initialize the scoreboard.

Child_init

Any handler registered for this hook is called once for each child process just after its creation (On the win32 platform, it is executed by the only child process before starting the worker threads). Here a module can perform tasks needed for every child process before starting request processing.

Child_exit (Apache 1.3 only)

Here each module can do work necessary to clean up before a child process exits. For example memory allocated for a child can now be released.

For more information about the multitasking server strategies, MPMs and master and child server tasks, take a look at section 4.3 in the next chapter.

create request (internal)

prepare all resources to read the request and send the response

post read request

This hook is triggered as last step upon reading a request.

Process a request and send a response Figure 3.10 shows how Apache processes an HTTP request. The special case of internal requests will not be explained further.

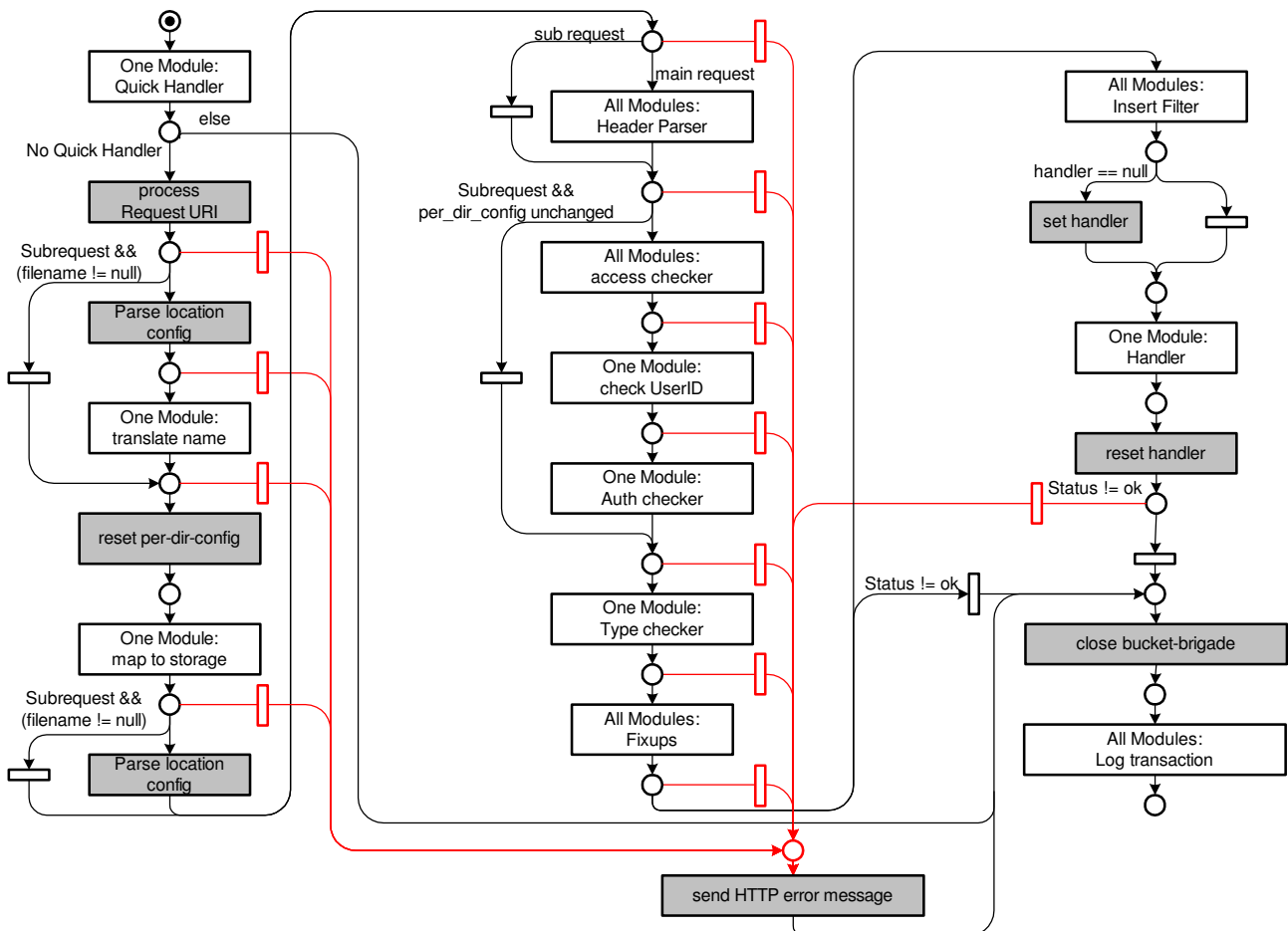


Figure 3.10: Apache request processing with module callbacks

quick handler

This hook is triggered before any request processing and can be used as shortcut (see top of figure 3.10). Cache modules can use this hook.

translate name

One module can translate the Request URI to a file name or to a different resource name.

map to storage (internal)

Determine the configuration for the requested resource, usually considering the “directory” and “files” sections. The proxy module registers a handler for this hook.



header parser

Here all modules can access the header and read module specific information, for example cookies. Since a similar task can be performed with the Post read request hook this one is not used in the standard Apache Modules.

access checker

This hook can be used to check whether the client machine may access the requested resource. Mainly this function is used to ex- or include specific IP address spaces or user agents (browsers). All modules are involved.

check_user_id

The hook is supposed to check whether the credentials supplied by the user (if any) are valid. Usually this means to look up user-name and password in a user database. Only one handler is allowed to perform this task.

auth checker

Here one module can check whether a user whose identity has been checked for a valid password in the preceding step, is authorized to access the resource he requested. Only one handler is allowed to complete this task.

type_checker

This hook allows one handler to determine or set the MIME type of the requested resource. The result has an impact on the selection of the content handler. The handler or even a filter may alter the MIME type of the response afterwards.

fixups

At this step all modules can have a last chance to modify the response header (for example set a cookie) before the calling the content handler

insert filter

This hook lets modules insert filters in the output filter chain.

handler

The hook for the content handler is the most important one in the request response loop: It generates or reads the requested resource and sends data of the response to the client using the output filter chain.

log transaction

Here each module gets a chance to log its messages after processing the request.

3.3.6 Inside a Module: mod_cgi

This module is discussed in detail to illustrate the structure of Apache Modules by a practical example.

Both distributions of Apache 1.3 and 2.0 include mod_cgi. This module is used to process CGI programs that can create dynamic web content. Due to the architectural differences between versions 1.3 and 2.0 discussed in the previous chapter the two versions of the module are different.

Mod_cgi in Apache 1.3

Module Info and referenced functions Usually the module info can be found at the end of the main source file for the specific module. In mod_cgi for Apache 1.3 the module info contains references to 2 handlers:

```
module MODULE_VAR_EXPORT cgi_module =
{
    STANDARD_MODULE_STUFF,
    NULL,                /* initializer */
    NULL,                /* dir config creator */
    NULL,                /* dir merger - default is to override */
    create_cgi_config,   /* server config */
    merge_cgi_config,    /* merge server config */
    cgi_cmds,            /* command table */
    cgi_handlers,        /* handlers */
    NULL,                /* filename translation */
    NULL,                /* check_user_id */
    NULL,                /* check auth */
    NULL,                /* check access */
    NULL,                /* type_checker */
    NULL,                /* fixups */
    NULL,                /* logger */
    NULL,                /* header parser */
    NULL,                /* child_init */
    NULL,                /* child_exit */
    NULL,                /* post read-request */
};
```

The first line within the module struct references a macro called "standard_module_stuff" which expands to the information each module has to provide. Two functions referenced in here are `create_cgi_config` and `merge_cgi_config`. The corresponding hooks for these handlers are `create server config` and `merge server config`. If you have a look at the two functions you will see that the first allocates and initializes memory for configuration data and the second merges the data stored for each virtual host with data stored for the master server.

The command table

```
static const command_rec cgi_cmds[] =
{
    {"ScriptLog", set_scriptlog, NULL, RSRC_CONF, TAKE1,
     "the name of a log for script debugging info"},
    {"ScriptLogLength", set_scriptlog_length, NULL, RSRC_CONF, TAKE1,
     "the maximum length (in bytes) of the script debug log"},
    {"ScriptLogBuffer", set_scriptlog_buffer, NULL, RSRC_CONF, TAKE1,
     "the maximum size (in bytes) to record of a POST request"},
    {NULL}
};
```

The references for command table and content handler do not point to functions but to structs. The command table struct contains references to the functions used to process the different directives that can be used for configuring mod_cgi. Within the command table each function is referenced with the additional keyword `TAKE1` which tells the core that only one parameter is accepted.



The content handlers

```
static const handler_rec cgi_handlers[] =
{
    {CGI_MAGIC_TYPE, cgi_handler},
    {"cgi-script", cgi_handler},
    {NULL}
};
```

The struct for the content handler registers the CGI mime-type as well as the "cgi-script" handler string with the function `cgi_handler`, which is the function called by the core for the content handler. Using that struct a module can register functions for more than one handler.

When the `type_checker` decided that the `mod_cgi` module should handle a request and then the core calls the content handler, it actually calls the function `cgi_handler`.

`cgi_handler` first prepares for executing a CGI by checking some pre conditions, like "Is a valid resource requested? ". Then it creates a child process by calling `ap_bspawn_child` that will execute the CGI program. Parameters for that function are among others the name of the function to be called within the process, here `cgi_child`, and a `child_stuff` struct that contains the whole request record. `Child_cgi` itself then prepares to execute the interpreter for the script and calls `ap_call_exec`, which is a routine that takes the different operating systems into account and uses the `exec` routines working for the currently used operating system. After that all output by the script is passed back to the calling functions until it reaches the `cgi_handler` handler function that then sends the data to the client including the necessary HTTP header.

Mod_cgi in Apache 2.0

Module Info and referenced functions In the version for Apache 2.0 the module info is much smaller. Most references to handlers for hooks are now replaced by the reference to the function `register_hooks`. All handlers except the handlers for the configuration management hooks are now dynamically registered using that function.

```
module AP_MODULE_DECLARE_DATA cgi_module =
{
    STANDARD20_MODULE_STUFF,
    NULL, /* dir config creator */
    NULL, /* dir merger --- default is to override */
    create_cgi_config, /* server config */
    merge_cgi_config, /* merge server config */
    cgi_cmds, /* command apr_table_t */
    register_hooks /* register hooks */
};
```

Even though syntax may vary, semantically the functions for configuration and the command table perform the same actions as in Apache 1.3. Having a look at the `register_hooks` function you can see an example how to influence the order to process the handler. While the `cgi_post_config` function shall be called absolutely first when the hook `post_config` is triggered, the `cgi_handler` should be called somewhere in the middle when the content handler hook is triggered.

```
static void register_hooks(apr_pool_t *p)
{
    static const char * const aszPre[] = { "mod_include.c", NULL };
    ap_hook_handler(cgi_handler, NULL, NULL, APR_HOOK_MIDDLE);
    ap_hook_post_config(cgi_post_config, aszPre, NULL, APR_HOOK_REALLY_FIRST);
}
```

Request handling employing filters In `mod_cgi` for Apache 2.0, the function `cgi_handler` is the start of the output filter chain. At first it behaves very much like its Apache 1.3 pendant. It prepares to start a process to execute the CGI program. It then retrieves the response data from that process. Most of the execution is done in the `cgi_child` function.

After it has got the response from the program, its task is to hand the just created brigade down the filter chain. That is done at the end of the function with a call to `ap_pass_brigade`. For example, it is now possible for a cgi program to output SSI (server-side includes) commands which are then processed by the include module. In that context the include module must have registered a filter that now gets the data from `mod_cgi`. Of course that depends on the configuration for the corresponding MIME type.

3.3.7 The Apache API

The Apache API summarizes all possibilities to change and enhance the functionality of the Apache web server. The whole server has been designed in a modular way so that extending functionality means creating a new module to plug into the server. The previous chapter covered the way in which modules should work when the server calls them. This chapter explains how modules can successfully complete their tasks.

Basically, all the server provides is a big set of functions that a module can call. These functions expect complex data structures as attributes and return complex data structures. These structures are defined in the Apache sources.

Again the available features differ between the two major Versions of Apache 1.3 and 2.0. Version 2.0 basically contains all features of 1.3 and additionally includes the Apache Portable Runtime (APR) which enhances and adds new functionality.

Memory management with Pools

Apache offers functions for a variety of tasks. One major service apache offers to its modules is memory management. Since memory management is a complex task in C and memory holes are the hardest to find bugs in a server, Apache takes care of freeing all used memory after a module has finished its tasks. To accomplish that, all memory has to be allocated by the apache core. Therefore memory is organized in pools. Each pool is associated with a task and has a corresponding lifetime. The main pools are the server, connection and request pool. The server pool lives until the server is shut down or restarted, the connection pool lives until the corresponding connection is closed and the request pool is created upon arrival and destroyed after finishing a request. Any module can request any type of memory from a pool. That way the core knows about all used memory. Once the pool has reached the end of its lifetime, the core deallocates all memory managed by the pool. If a module



needs memory that should even have a shorter lifetime than any of the available pools a module can ask Apache to create a sub pool. The module can then use that pool like any other. After the pool has served its purpose, the module can ask Apache to destroy the pool. The advantage is that if a module forgets to destroy the sub pool, the core only has to destroy the parent pool to destroy all sub pools.

Additionally Apache offers to take care of Array and Table management, which again makes memory management easier. Arrays in Apache can grow in size over time and thus correspond to Vectors in Java and Tables contain key/value pairs and thus are similar to hash tables. Section 4.6 provides further informations about the pool technology.

Data types

Apache offers a variety of functions that require special parameters and return values of special structure. Therefore it is essential to know about the Data types used by the Apache API. Most fields contained in any of these records should not be changed directly by a module. Apache offers API functions to manipulate those values. These functions take necessary precautions to prevent errors and ensure compatibility in later versions.

`request_rec`

This is the most important data structure within Apache. It is passed to the module's handlers during the request-response phase. It contains all information about the request as well as references to information and configuration data of the main server and virtual host (`server_rec`) and information about the connection the request belongs to (`connection_rec`). Each module can find the reference to the memory pool for that request and to a variety of information that has been gathered about that request so far. It also contains a structure that contains various different formats of the URI from URI translation phase.

The name is somehow misleading as this data structure is also used to gather data for the response, especially header fields.

`server_rec`

This structure contains information about the server, especially configuration. Apache keeps these structures for the main server and for each virtual host. Based upon the request a module handles, the core passes the corresponding server structure to the module. The structure itself contains fields with information like server name and port, timeout and keep-alive settings. Via the `server_rec` a module can also access its own server-based configuration directives.

`connection_rec`

This structure contains information about the current connection. With HTTP 1.1 multiple requests can be submitted via one connection. Therefore a connection can exist longer than one request as long as the server and the client support persistent connections. Since the `connection_rec` also contains a memory pool, any module dealing with a specific request can store data that is persistent during one connection. The `connection_rec` can be used to determine the main server if the connection was made to a virtual server. Various user data is also stored here.

API Functions

Besides memory management Apache assists the modules in various ways. The main target of the API is full abstraction from the operating system and server configuration. Apache offers functions for manipulating Apache data structures and can take precautions the module does not need to know about. The core can also perform system calls on behalf of the module and will use the routines corresponding to the operating system currently in use. That way each module can be used in any operating systems environment. For example the Apache API includes functions for creating processes, opening communication channels to external processes and sending data to the client. Additionally Apache offers functions for common tasks like string parsing.

Within the Apache API, functions can be classified into the following groups:

- Memory Management
- Multitasking (Processes only)
- Array Manipulation
- Table Manipulation
- String Manipulation (e.g. parsing, URI, Cookies)
- Network I/O (Client Communication only)
- Dynamic Linking
- Logging
- Mutex for files
- Identification, Authorization, Authentication

For further information on the Apache 1.3 API and how to write Apache Modules see [4].

Apache 2.0 and the Apache Portable Runtime (APR)

Version 2.0 of Apache introduces the Apache Portable Runtime, which adds and enhances functionality to the Apache API. Due to the APR Apache can be considered a universal network server that could be enhanced to almost any imaginable functionality. That includes the following platform independent features:

- File I/O + Pipes
- Enhanced Memory Management
- Mutex + Locks, Asynchronous Signals
- Network I/O
- Multitasking (Threads & Processes)



- Dynamic Linking (DSO)
- Time
- Authentication

A goal of the APR is to set up a common framework for network servers. Any request processing server could be implemented using the Apache core and the APR.

Chapter 4

Inside Apache

4.1 Introduction

This chapter focuses on the implementation of Apache. You should only read it if you

- are interested in the internal structure of a network server
- intend to work on or change Apache

Some sections contain descriptions related closely to the source code. We analyzed Apache version 1.3.17 in the project's first year and compared these results to Apache 2.0 during the second year. The online version of this document provides links to locations in the source code of both versions.

You should know the concepts of an HTTP server shown in chapter 2 and be familiar with the characteristics of Apache depicted in chapter 3. Furthermore you should be able to read C code.

Section 4.2 illustrates the first step of a source analysis: You need to figure out the structure of the Apache source distribution.

A basic concept of a network server is the way it handles concurrency. Section 4.3 at first introduces a common way to implement a multitasking server and then compares it with the way Apache handles concurrency.

Figure 4.8 gives an overview of the behavior of the server and introduces the loops the further sections focus on. It also covers the server's reaction to the control commands of the administrator (restart or end).

The master server loop shown in section 4.3.3 is closely related to the multitasking model of Apache.

The request-response loop in section 4.4 forms the heart of every HTTP server and is therefore similar to the behavior of a simple HTTP server shown in figure 2.2. In this section you will also find the "other side" of the module callbacks explained in section 3.3.

The further sections of this chapter deal with implementation details of various Apache concepts.



4.2 Structure of the Source Distribution

4.2.1 Apache 1.3.17 source distribution

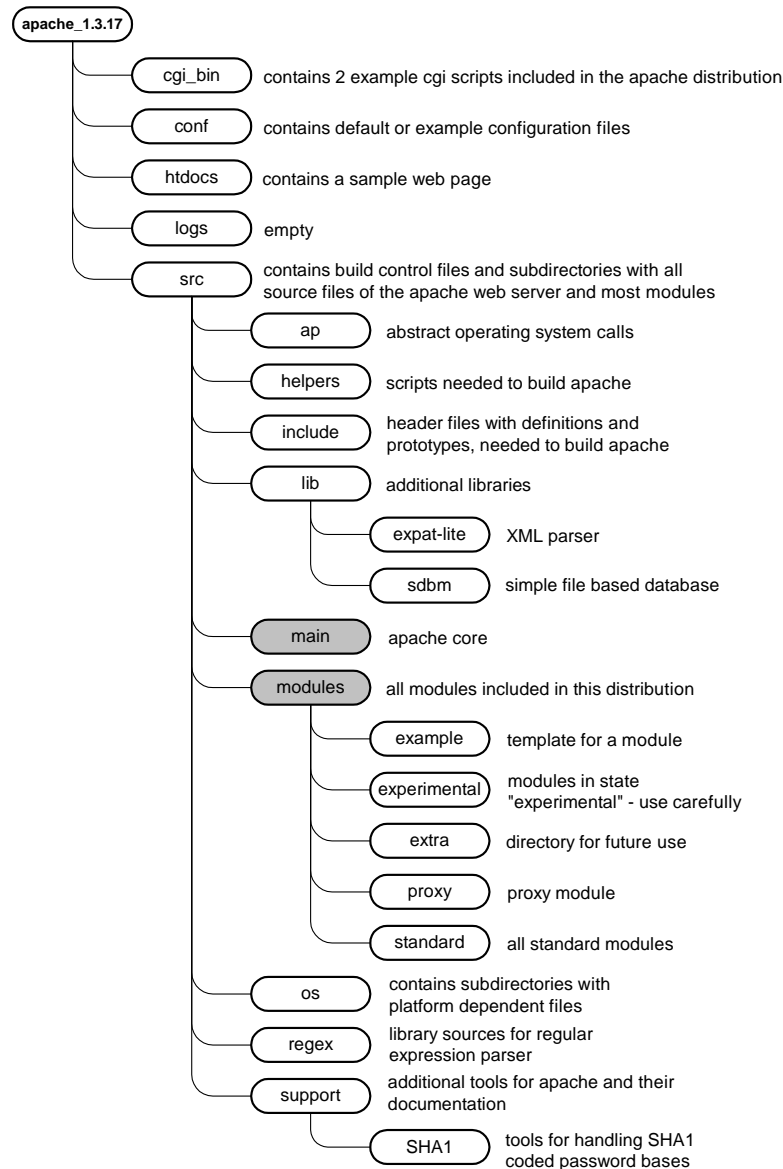


Figure 4.1: Directory structure of the Apache HTTP Server 1.3.17 source distribution

The source distribution of Apache version 1.3.17 contains 780 files in 44 subdirectories; 235 files contain C source code. Figure 4.1 shows the directory structure of the Apache 1.3.17 source distribution. The most important directory is `src`. It contains subdirectories with all source files of the Apache web server and most modules. For us, the most interesting subdirectories are `main` and `modules` as they contain the central source code which is needed to understand how Apache works.

For more details, browse through the on-line document “Structure of the Apache HTTP Server 1.3.17 source distribution” (see apache.hpi.uni-potsdam.de/sources)

4.2.2 Apache 2.0.45 source distribution

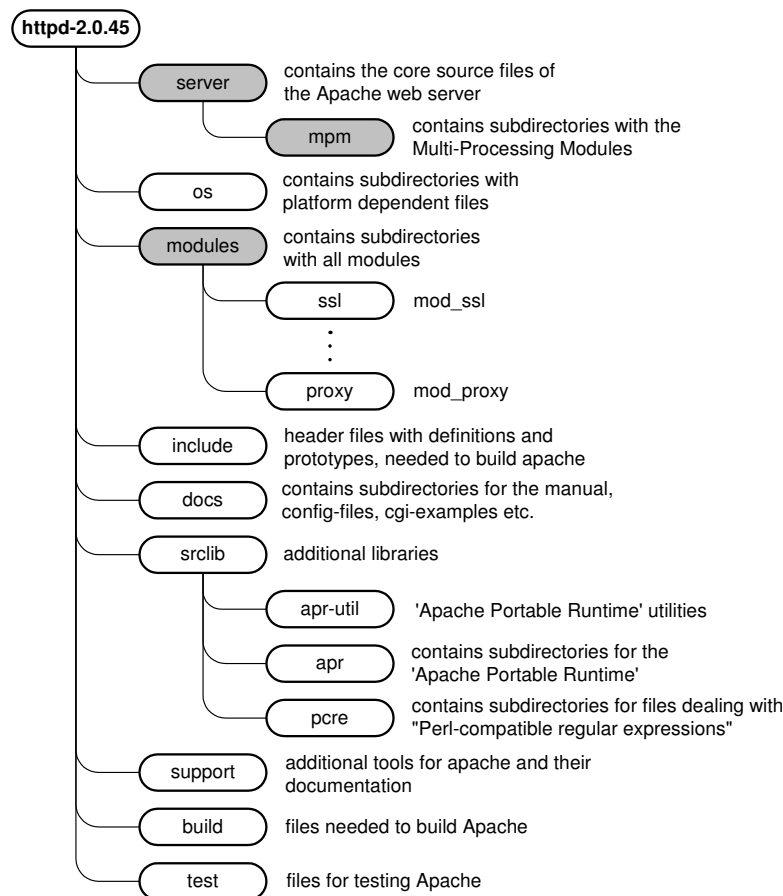


Figure 4.2: Directory structure of the Apache HTTP Server 2.0.45 source distribution

Figure 4.2 shows the directory structure of the Apache 2.0.45 source distribution.

Compared to Version 1.3, Apache 2 is much bigger. The Apache 2.0.45 distribution includes 2165 files in 183 subdirectories; 704 files contain C source code (appr. 280 000 lines incl. comments). The directory structure of the source distribution has also changed significantly.

Nonetheless, the parts important for us are still located in a few central directories. The core files are now located in the directory *server*, with the MPMs located in the subdirectory *mpm*. The directory *module* still contains subdirectories with all modules included in the distribution.

More details can be found browsing the on-line document "Structure of the Apache HTTP Server 2.0.45 source distribution" (see apache.hpi.uni-potsdam.de/sources)

4.3 Multitasking server architectures

The most influencing factor when creating multitasking architectures is the operating system in use. With every operating system the environment and their performance attributes change. Therefore certain multitasking architectures are incompatible or not performant



enough on certain operating systems. The second major influencing factor are the use-scenarios. Depending on how much processing is involved with a single request, how many requests a server will have to handle and/or whether requests logically depend on each other, certain architectures might be more advantageous than others.

Section 4.3.1 explains how a common multitasking network server architecture works and discusses its shortcomings if used as an HTTP server. The Apache server architecture will be shown in section 4.3.2.

4.3.1 Inetd: A common multitasking architecture

Master server as a gatekeeper

Network servers handling concurrent requests usually show a multitasking architecture. Although Apache doesn't use it, we will now describe a common pattern using processes which works as follows:

A *master server* process is waiting for incoming requests. Whenever a request from a client comes in, the master server establishes the connection and then creates a *child server* process by forking itself. The child server process handles the request(s) of the client while the master server process returns to its waiting state.

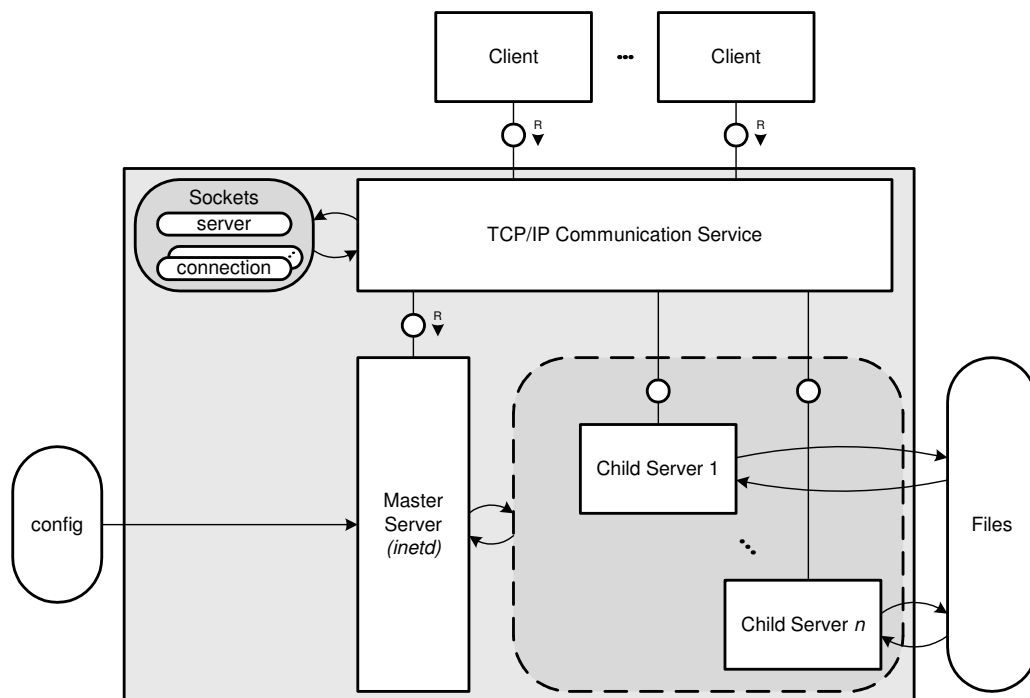


Figure 4.3: Multiprocessing Architecture of an inetd server

In figure 4.3 you see the structure of this kind of multiprocessing server. At the top there are one or many clients sending requests (R) to the server. The requests are received by the TCP/IP Communication Service of the operating system. The Master Server has registered itself as responsible for any request that comes in. Therefore the communication service

wakes it up. The Master Server accepts the connection request so the communication service can establish the TCP/IP connection and create a new socket data structure.

The master server creates a child server process by doing a `fork()` system call. In figure 4.3 this is symbolized by the “write” arrow from the master server to the storage area enclosed by a dashed line. The child server knows about the connection because it knows the connection socket. It can now communicate with the client until one of them closes the connection. Meanwhile the master server waits for new requests to arrive.

With TCP/IP, a connection end point has to be identified by the IP address of the machine and a port number (for example, the port for HTTP requests has number 80). The master server process registers itself as a listener for the port (which in turn becomes a server port). Note that a connection request arrives at the server port while the connection is established using a connection port. The major difference between a server and a connection port is that a server port is solely used to accept connections from any client. A connection port uses the same TCP/IP portnumber but is associated with one specific connection and therefore with one communication partner. Connection ports are used to transmit data and therefore the server port can remain open for further connection requests.

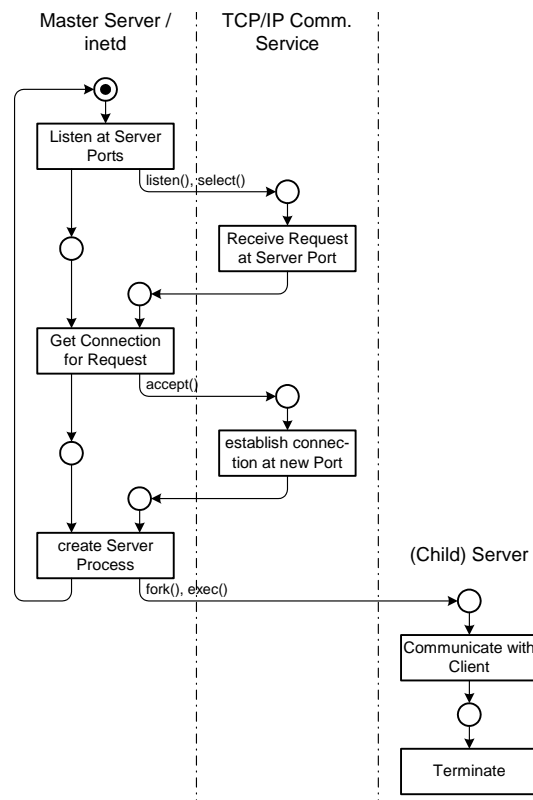


Figure 4.4: Behavior of a multiprocessing server

The behavior of the server is shown in figure 4.4. The system calls `accept()` or `select()` block¹ the master server process until a request comes in. `accept()` waits for requests on one server port while `select()` is a means to observe multiple ports. In this case, after a request has been received by the TCP/IP Communication Service, the master server can establish the connection with the system call `accept()`. After that it creates a new process

¹this means the server process remains inactive until a request comes in



with `fork()`. If the request has to be handled by a different program, it has to be loaded and executed with `exec()`.

The INETD

The INETD is a typical server using the multitasking architecture described above. It waits for requests on a set of ports defined in the configuration file `/etc/inetd.conf`. Whenever a request comes in, `inetd` starts the (child) server program defined in the configuration file. That program then handles the request.

Apache also provides a mode of operation to work with the `inetd`. In this case, the `inetd` is the gatekeeper for the HTTP port (80) and starts Apache whenever an HTTP request arrives. Apache answers the request and exits.

Drawbacks

This multiprocessing architecture is useful if the handling of the client request takes some time or a session state has to be kept by the (child) server because the communication between client and server does not end after the response to the first request.

HTTP, however, is a stateless protocol. No session information needs to be kept by the server — it only needs to respond to one request and can “forget” about it afterwards. An HTTP server based on the `inetd` architecture would be inefficient. The master server would have to create a process for each HTTP connection, which would handle this one connection only and then die. While the master server creates a process it cannot accept incoming requests. Although process creation does not take a long time on modern operating systems, this gatekeeper function of the master server forms a bottleneck for the entire server.

4.3.2 Overview — Apache Multitasking Architectures

All Apache Multitasking Architectures are based on a task pool architecture. At start-up, Apache creates a number of tasks (processes and/or threads), most of them are idle. A request will be processed by an idle task, therefore there’s no need to create a task for request processing like the `inetd` described in section 4.3.1.

Another common component is the master server, a task controlling the task pool — either control the number of idle tasks or just restart the server something goes wrong. The master server also offers the control interface for the administrator.

In the following, the preforking architecture will be presented as the first and most important architecture for unix systems. Then we present a selection of other Apache multitasking architectures and emphasize the differences concerning the preforking architecture.

4.3.3 The Preforking Multiprocessing Architecture

The leader–followers pattern

The preforking architecture is based on a pool of tasks (processes or threads) which are playing 3 different roles:

- wait for requests (listener)
- process a request (worker)
- queue in and wait to become the listener (idle worker)

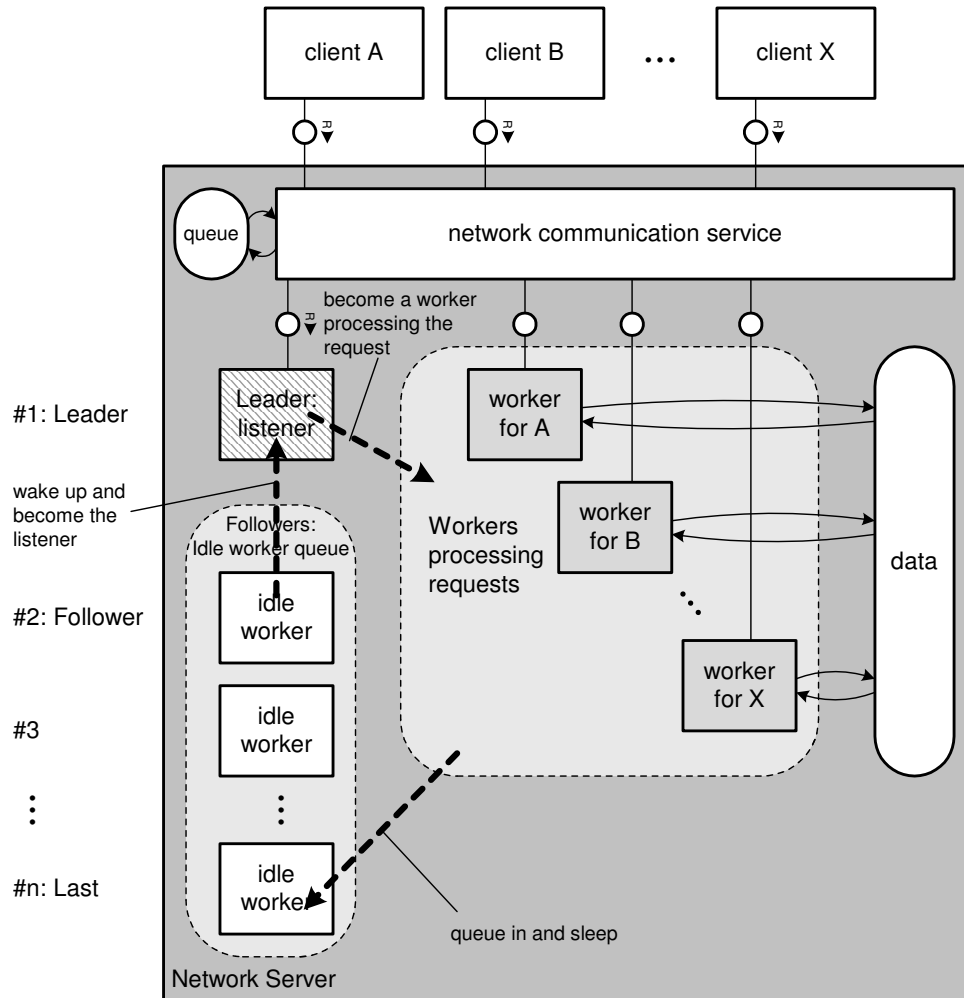


Figure 4.5: The leader–followers pattern used in the preforking server architecture

A description of the pattern can be found in [5]. Figure 4.5 shows the structure of the system: The listener is the leader. Only one task can be granted the right to wait for connection requests. If the listener gets a request, it hands over the right to listen and switches his role to worker, that means he processes the request using the connection he established as listener. If he's finished processing the request, he closes the connection and becomes an idle worker. That means he queues in waiting to become the listener. Usually an idle worker task will be suspended.

What are the differences between the server strategy described in section 4.3.1 and the leader–follower strategy? Firstly, an incoming request will be treated immediately by the listener task — no new task has to be created. On the other hand there should always be a certain number of idle worker tasks to make sure there is always a listener. Secondly, there is no need to pass information about a request to another task because the listener just switches his role and keeps the information.



The task pool must be created at server start. The number of tasks in the pool should be big enough to ensure quick server response, but a machine has resource restriction. The solution is to control the number of tasks in the pool by another agent: the master server.

Preforking Architecture

The Preforking architecture was the first multitasking architecture of Apache. In Apache 2.0 it is still the default MPM for Unix. The Network MPM very closely resembles the Preforking functionality with the exception that it uses Network threads instead of unix processes. Summarizingly the Preforking architecture of Apache takes a conventional Approach as each child server is a process by itself. That makes Preforking a stable architecture but also reduces performance.

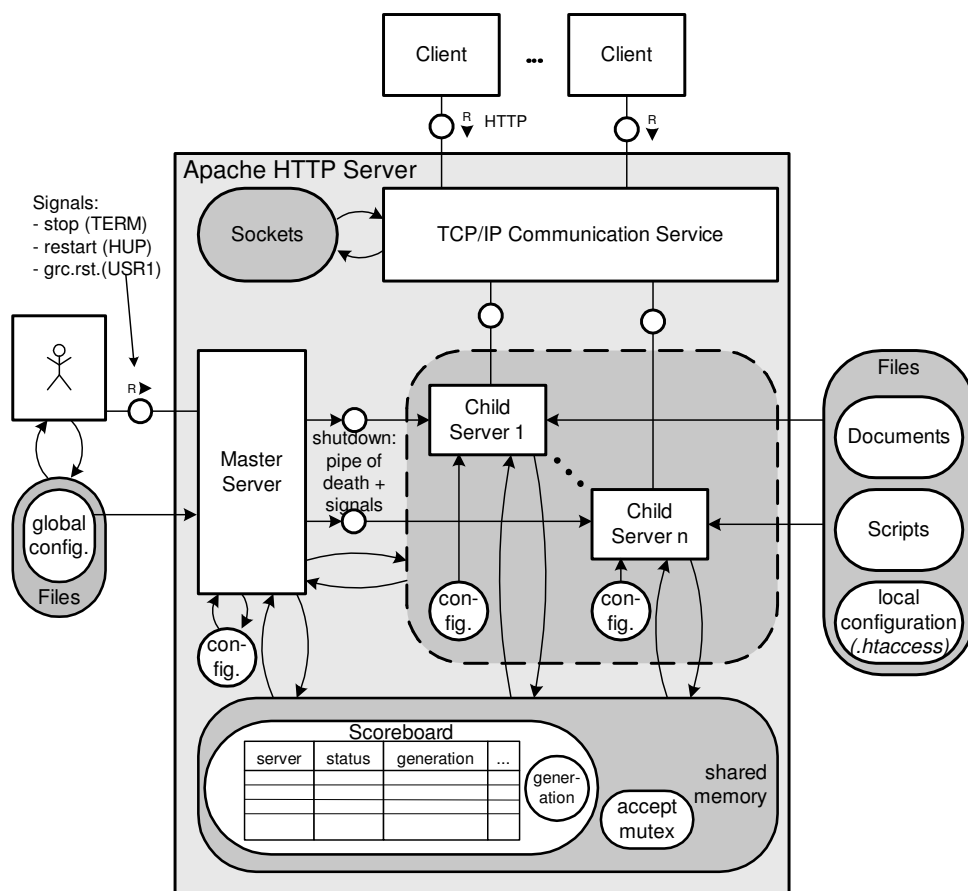


Figure 4.6: The Apache 2.0 Preforking MPM

The structure diagram in figure 4.6 shows the structure of the Preforking architecture of Apache 2.0 and is important for the description of the behavior of Apache. You can see which component is able to communicate with which other component and which storage a component can read or modify. The block diagram for the Apache 2.0 MPM version of the Preforking architecture very much resembles the version that was used on Apache 1.3, however there is one difference: The Master Server uses a “pipe of death” instead of signals to shut down the child servers for a (graceful) restart.

The Preforking architecture shown in figure 4.6 seems to be very similar to the `inetd` architecture in figure 4.3 at first sight. There is one master server and multiple child servers. One big difference is the fact that the child server processes exist *before* a request comes in. As the master server uses the `fork()` system call to create processes and does this before the first request comes in, it is called a *preforking* server. The master server doesn't wait for incoming requests at all — the existing child servers wait and then handle the request directly.

The master server creates a set of idle child server processes, which register with the TCP/IP communication service to get the next request. The first child server getting a connection handles the request, sends the response and waits for the next request. The master server adjusts the number of idle child server processes within given bounds.

General Behavior

Figure 4.7 shows the overall behavior of the server, including the master server and the child servers.

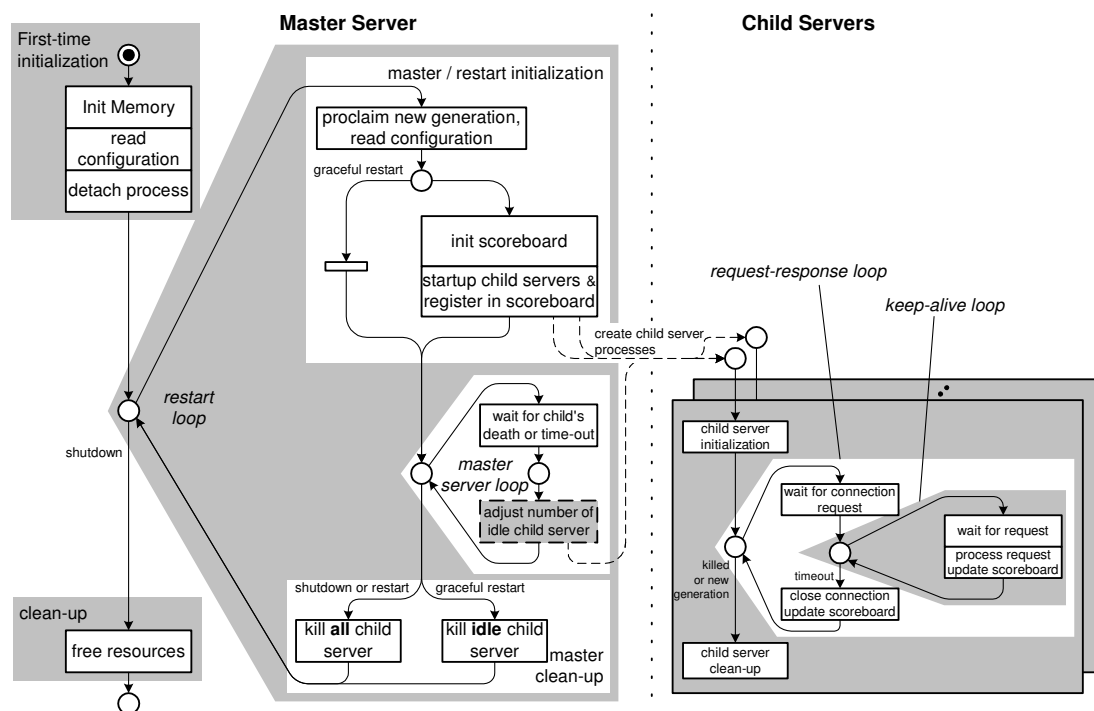


Figure 4.7: Overview: The behavior of Apache

Independently from the underlying multitasking architecture, Apache's behaviour consists of a sequence of the following parts which will be discussed individually for each of the architectures:

- **First-time initialization:**
Allocate resources, read and check configuration, become a daemon.
- **The restart loop:**
(Re-)read configuration, create task pool by starting child server processes and enter the master server loop.



- The master server loop:
Control the number of idle child server processes in the task pool.
- The request–response loop (Child server only):
Wait to become leader, wait for a connection request, become worker and enter the keep–alive–loop.
- The keep–alive–loop (Child server only):
Process HTTP requests
- Clean–up before deactivation (Master server and Child servers)

Figure 4.8 shows the behavior of Apache using the preforking architecture in greater detail. As each multitasking architecture distinguishes itself from others by using different means to create and organize child servers, the behaviour of different multitasking architectures mainly differs when child servers, also called workers are created during the restart loop and within the master server loop when the workers are monitored and replaced.

Initialization & Restart Loop

Initialization The server structure shown in figure 4.6 has to be set up at start–up (start processes, create sockets and pipes, allocate memory) and destroyed at shutdown. This is called activation and deactivation.

There are three types of initializations:

1. at the first activation of Apache
2. every time the restart loop is run
3. every time a child server is created

Apache 2.0 starts with `main()`. After entering the Restart Loop, it calls the configured MPM using `ap_mpm_run()`. (Apache 1.3 using Preforking starts with the procedure `REALMAIN()`.) The following comments explain the operations shown in figure 4.8:

- create static pools: Apache initializes memory areas in its own memory management (pool management, see section 4.6)
- register information about prelinked modules: The biggest part of the HTTP server functionality is located in the modules (see section 3.3 for further details). Modules can either be included in the apache binary or loaded dynamically. Even if they are included in the binary (prelinked), they have to be registered.
- read command line and set parameters: The administrator can override defaults or config file configuration data with command line parameters. The command line parameter `-X` enforces the ‘register one process mode’ and can be used for debugging purposes. It prevents the creation of child server processes. If no child server processes exist, there is no need for a master server — the one existing process enters the request–response loop and behaves like a single child server.

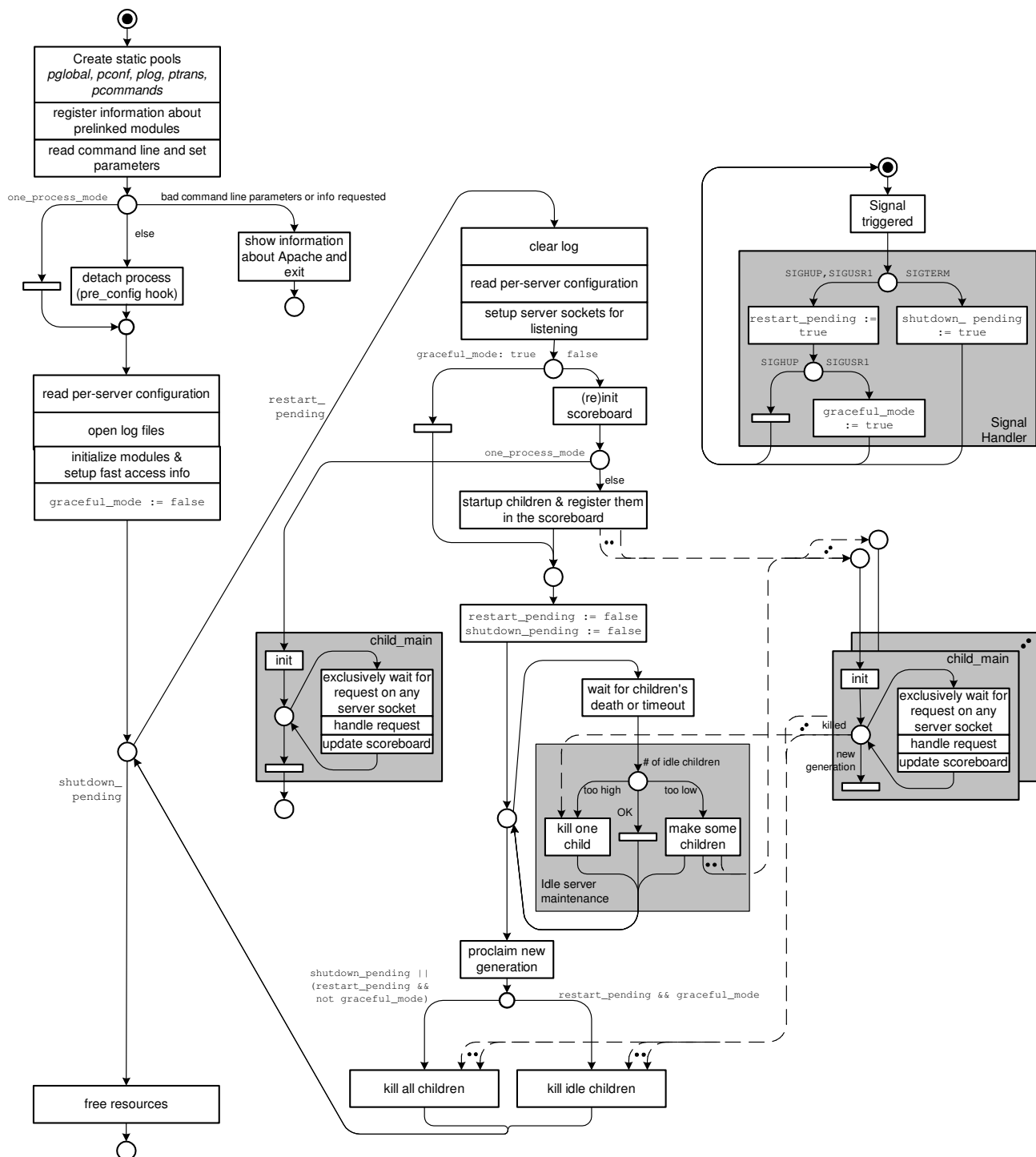


Figure 4.8: Details of the behavior of Apache



- **read per-server configuration:** The master server (nothing else exists at this time) reads the configuration files and merges the information with its configuration data. Configuration data also includes information about the modules to be loaded. Note that configuration data has to be read a second time in the restart loop!
“per-server configuration” means all static configuration data in contrast to the configuration data in `.htaccess` files called “per-request configuration”.
- **graceful_mode := false:** At this time only the master server process exists, so there is no sense in using graceful mode. (In graceful mode — see section 4.3.3 — Apache performs a restart keeping active child servers alive.)
- **detach process:** Each process is usually a child process of the process that created it. The parent process can be the shell process, for example. If the shell terminates, all child processes of the shell process are terminated, too. Furthermore, all input and output streams of the child process (STDIN, STDOUT, STDERR) are connected with the shell.
Apache performs the detach after it has read the configuration data and tried to initialize the modules. After the detach no error message will be printed in the shell, because the master server has disconnected from the shell and now runs as a background task. The detach process consists of the following actions:
 1. create a clone of the current process using `fork()`
 2. immediately stop the parent process
 3. the clone detaches the input and output streams from the shell
 4. the clone sets its process group ID to the group ID of process number 1 (init). It ‘denies’ every relationship with its ‘real’ parent process and from now on only depends on the system process init.

The ‘one_process_mode’ is useful for debugging purposes. Apache skips the detach operation and is still available for the debugger.

Restart Loop Every time the administrator forces a restart of the Apache server, it processes the *restart loop* which can be found in `main()`. (In Apache 1.3, the restart loop is located in the procedure `standalone_main()`.) After reading the configuration, it calls `ap_mpm_run()` of the Preforking MPM.

The loop has the following parts:

1. initialize and prepare resources for new child servers, read and process configuration files
2. create child servers
3. Master server: observe child servers (Master Server Loop, see section 4.3.3).
Child servers: Handle HTTP requests (Request-Response Loop, see section 4.4).
4. kill child servers (graceful restart: kill idle child servers only)

The configuration files are read by the master server only. The child servers get their configuration data when they are created by the master server. Whenever the administrator wants to apply configuration changes to the server, he has to enforce a restart. A graceful restart allows child server processes to complete their processing of current requests before being replaced by servers of the new generation. Each child server updates its status in the scoreboard (a shared memory area) and compares its own generation ID with the global generation ID, whenever it completes request handling.

- **read per-server configuration:** The master server reads and processes the configuration files. At this time only the master server (and maybe some non-idle child servers of the old generation) exist.
- **set up server sockets for listening:** Apache can listen on many ports. It is important not to close the server sockets during restart.
- **init scoreboard:** In case a graceful restart is processed, the scoreboard entries for the remaining child servers must be kept. Otherwise there are no child servers and the scoreboard can be initialized.
- **one_process_mode:** This mode is used for debugging purposes (see also `detach`). The master server becomes child server and enters the request-response loop.
- **startup children & register them in the scoreboard:** The master server creates child server processes with the procedure `startup_children()`. It uses the `fork()` system call. As a consequence, all child server processes get a copy of the memory imprint of the master server and its system resources. Therefore they “know” the configuration data and have access to the TCP/IP sockets and the log file sockets.
If Apache is started by the super user (root), the master server process is the only process using the root User ID. The child server processes initialize, set their user ID to a non-privileged account like “nobody” or “wwwrun” and enter the request-response loop.
The master server creates an entry in the scoreboard for every child server including its process ID and generation ID.
- **restart/shutdown pending := false:** The master server enters the master server loop. It exits the loop only if one of those global variables is set to “true” by the signal handler.
- **Master server loop:** (see section 4.3.3 and figure 4.9) At the beginning of the loop the master server waits a certain time or receives the notification that a child server has died. Then it counts the number of idle child servers and regulates their number by creating or killing one.
- **proclaim new generation:** Each time the master server processes the restart loop, it increments the generation ID. All child servers it creates have this generation ID in their scoreboard entry. Whenever a child server completes the handling of a request, it checks its generation ID against the global generation ID. If they don’t match, it exits the request-response loop and terminates.
This behavior is important for the graceful restart.



- **finish all/idle children:** Both shutdown and restart result in the death of all child servers. When the administrator requests a graceful restart, only the idle child servers are killed.
- **free resources:** Apache returns the occupied resources to the system: Memory, TCP/IP ports and file handles.

Inter-Process Communication (Signals and Pipe of Death)

Apache is controlled by signals. Signals are a kind of software interrupts. They can occur at any time during program execution. The processor stops normal program execution and processes the signal handler procedure. If none is defined in the current program, the default handler is used which usually terminates the current program. After the execution of the signal handler the processor returns to normal execution unless the program was terminated by the signal.

Administrator controls the master server The administrator can send signals directly (using `kill` at the shell command prompt) or with the help of a script. The master server reacts to three signals:

- **SIGTERM:** shut down server (set `shutdown_pending := true`)
- **SIGHUP:** restart server (set `restart_pending := true` and `graceful_mode := false`)
- **SIGUSR1:** restart server gracefully (set `restart_pending := true` and `graceful_mode := true`)

The signal handlers for the master server are registered in the procedure `set_signals()`. It registers the signal handler procedures `sig_term()` and `restart()`.

In the upper right corner of figure 4.8 you see a small petri net describing the behavior of the signal handler of the master server. Apache activates and deactivates signal handling at certain points in initialization and in the restart loop. This is not shown in figure 4.8.

Master Server controls the child servers While the administrator controls the master server by signals only, the master server uses either signals or a pipe to control the number of child servers, the Pipe of Death. (Apache 1.3 used signals only).

For a shutdown or non-graceful restart, the master server sends a SIGHUP signal to the process group. The operating system “distributes” the signals to all child processes belonging to the group (all child processes created by the master server process). The master server then “reclaims” the notification about the termination of all child servers. If not all child processes have terminated yet, it uses increasingly stronger means to terminate the processes.

A graceful restart should affect primarily the idle child server processes. While Apache 1.3 just sent a SIGUSR1 signal² to the process group, Apache 2 puts “Char of Death” items

²The Apache 1.3 child server’s reaction to a SIGUSR1 signal: Terminate if idle else set `deferred_die` and terminate later. (See signal handler `usr1_handler()` registered at child server initialization.)

normal restart	graceful restart
send SIGHUP to the process group	use the Pipe of death to terminate the idle child servers (Apache 1.3: send SIGUSR1 to the process group)
all child servers terminate	idle child servers receive an item through the Pipe of Death and terminate, busy child servers either receive a job through the Pipe of death later, or check the generation in the scoreboard, set <code>die_now=true</code> and terminate later (Apache 1.3: Idle child servers terminate while the busy ones set <code>deferred_die := true</code> and terminate later.)
<code>reclaim_child_processes</code> : gather termination notification for all child processes	— (see Master server loop)
initialize scoreboard	keep scoreboard
startup children	—
Master server loop: Enter normal operation	Master server loop: Create a child server for every termination notification

Table 4.1: Normal versus graceful restart

into the Pipe of Death (pod). The busy child servers will check the pipe after processing a request, even before comparing their generation. In both cases they set the `die_now` flag and terminate upon beginning a new iteration in the request–response loop

Table 4.1 lists the differences between a normal and a Graceful Restart .

Pipe of Death (PoD) The Master server of Apache 2.0 uses the Pipe of Death for inter-process communication with the child servers to terminate supernumerary ones and during graceful restart. All child servers of one generation share a pipe.

If the master server puts a Char of Death in the queue using `ap_mpm_pod_signal()` or sends CoD to all child servers with `ap_mpm_pod_killpg()`, these procedures also create a connection request to the listening port using the procedure `dummy_connection()` and terminate the connection immediately. The child server waiting for new incoming connection requests (the listener) will accept the request and skip processing the request as the connection is already terminated by the client. After that it checks the PoD which causes him to terminate. Busy child servers can continue serving their current connection without being interrupted.

Master Server Loop

Overview In this loop the master server on the one hand controls the number of idle child servers, on the other hand replaces the child servers it just killed while performing a graceful restart.

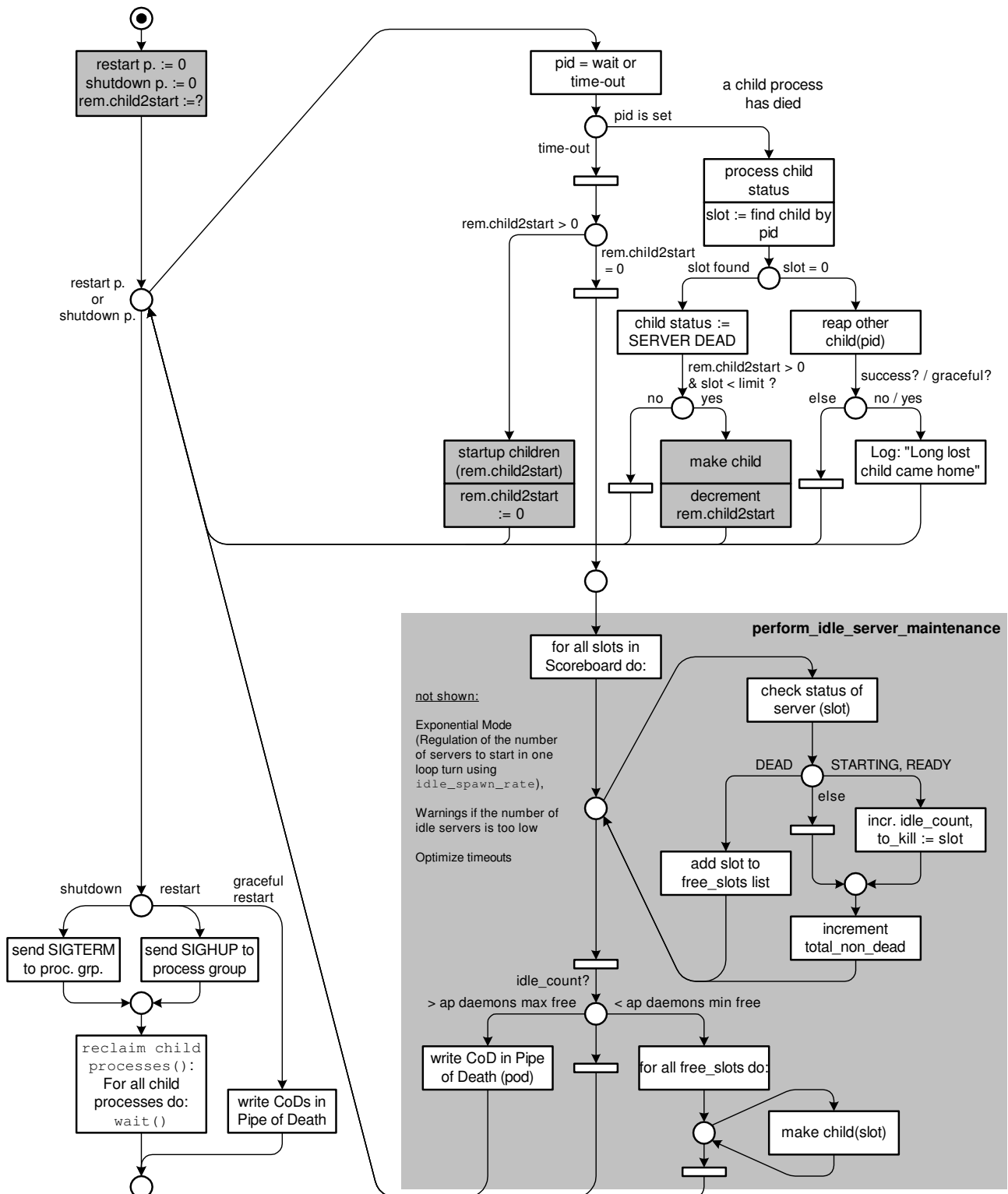


Figure 4.9: The Master Server Loop

While the restart loop can be found within the server core in `main()`, the master server loop it is located within the corresponding MPM, which in this case is the Preforking: `ap_mpm_run()`. (In Apache 1.3 it can be found in the procedure `standalone_main()` in the file `http_main.c`.)

In figure 4.9 you see the details of the loop. The upper part deals with the reaction to the death of a child process and special issues of a graceful restart. The lower part is labeled “perform idle server maintenance”. It shows a loop in which the master server counts the number of idle servers and gets a list of free entries in the scoreboard. It compares the number of idle children (`idle_count`) with the limits given in the configuration (`ap_daemons_max_free` and `ap_daemons_min_free`). If there are too many idle servers, it kills exactly one of them (the last idle server in the scoreboard). If the number of idle child servers is too low, the master server creates as many child server processes as needed (see exponential mode below).

Graceful Restart — Reaction to the death of a child process The following remarks mainly deal with the graceful restart and the reaction to the death of a child process:

- `set rem.child2start`: The variable `remaining_children_to_start` is used only in the case of a graceful restart. It holds the number of child servers that should be started after the server is up. Notice that the master server does not use the `startup_children` procedure. While performing the `wait()` system call, the master server gets a notification for every idle child server it has killed. If the number of initial child servers was changed in the configuration, it would be wrong to just replace dead children, so the master server uses `remaining_children_to_start` to control the number.
- `pid := wait or timeout`: The `wait()` system call is used to wait for the termination of a child process created with `fork()`. After waiting for a given period of time, the master server continues execution of the master server loop even if it has not received a termination notification.
- `wait()` was successful (`pid` is set):
 - `process_child_status`: Get the reason for the death of the child process
 - `find_child_by_pid`: Look for the scoreboard entry
 - `entry (slot) found`: set child status to `SERVER DEAD`. If `remaining_children_to_start` is not zero, create a new child server to replace the dead child server.
 - `entry not found`: Check if this child process has been an “other child”³(`reap_other_child()`, see below). If it is neither an “other child”

³In some cases the master server has to create child processes that are not child server processes. They are registered as “other child” in a separate list. An example: Instead of writing log data to files, Apache can stream the data to a given program. The master server has to create a process for the program and connect its STDIN stream with its logging stream. This process is an “other child”. Whenever the server is restarted, the logging process gets a `SIGHUP` or `SIGUSR1`. Either it terminates and has to be re-created by the according module (the logging module in the example) or it stays alive. The module must check the “other child” list to find out if it has to create or re-use a process.



nor a scoreboard entry matches, and if graceful mode is set, then the following situation must have happened:

The administrator has reduced the number of allowed child servers and forced a graceful restart. A child server process that has been busy had a slot greater than the allowed number. Now it terminates, but its entry can not be found in the scoreboard.

- **time-out:** If `remaining_children_to_start` is still not zero after all terminated child servers have been replaced, there are more servers to be created. This is done by the procedure `startup_children()`.

Performing Idle Server Maintenance The lower part of figure 4.9 shows the behavior of the procedure `perform_idle_server_maintenance()` which is called whenever a time-out occurred and the graceful restart has been finished.

The master server counts the number of idle servers and the number of remaining slots (entries) in the scoreboard and compares it with three limits given in the configuration:

`ap_daemons_limit` maximum number of child servers. The sum of busy and idle child servers and free slots or just the number of slots of the scoreboard.

`ap_daemons_max_free` maximum number of idle child servers. If the number of idle servers is too high, one child server process is killed in each loop.

`ap_daemons_min_free` minimum number of idle child servers. If the number of idle servers is too low, the master server has to create new child servers. If no free slots are left (because `ap_daemons_limit` has been reached), a warning is written to the log.

Exponential mode: Some operating systems may slow down if too many child processes are created within a short period. Therefore the master server does not immediately create the needed number of child servers with `make_child()`. It creates one in the first loop, two in the second, four in the third and so on. It holds the number of child servers to be created in the next loop in the variable `idle_spawn_rate` and increments it with every turn until the number of idle child servers is within the limit.

Example: `ap_daemons_min_free` is set to 5 but suddenly there is only 1 idle server left. The master server creates one child server and waits again. 2 idle servers are still not enough, so the master creates 2 more child servers and waits again. In the meantime, a new request occupies one of the new child servers. The master server now counts 3 idle child servers and creates 4 new ones. After the time-out it counts 7 idle child servers and resets the `idle_spawn_rate` to 1.

Child Servers (Worker)

The Child Servers sometimes referred to as a workers form the heart of the HTTP Server as they are responsible for handling requests. While the multitasking architecture is not responsible for handling requests it is still responsible for creating child servers, initializing them, maintaining them and relaying incoming connections to them.

Initialization, Configuration and Server restarts The master server creates child server processes using the `fork()` system call. Processes have separate memory areas and are not allowed to read or write into another processes' memory. It is a good idea to process the configuration once by the master server than by each child server. The configuration could be stored in a shared memory area which could be read by every child server process. As not every platform offers shared memory, the master server processes the configuration files before it creates the child server processes. The child server processes are clones of the master server process and therefore have the same configuration information which they never change.

Whenever the administrator wants to apply changes to the server configuration, he has to advice the master server to read and process the new configuration data. The existing child server processes have the old configuration and must be replaced by new processes. To avoid interrupting the processing of HTTP requests, Apache offers the "graceful restart" mode (see section 4.3.3), which allows child servers to use the old configuration until they have finished processing their request.

The initialization of a child server can be found in the corresponding MPM (Preforking: `child_main()`, Apache 1.3: `child_main()`). It consists of the following steps (see also figure 4.13):

- establish access to resources: The child server process has just been created by the master server using `fork()`. At this time the child server process has the same privileges as the master. This is important if Apache has been started by the super user (root). Before the child server sets its user ID to a non-privileged user, it must get access to common resources:
 - Private memory (`ap_make_sub_pool()`)
 - The scoreboard (may be implemented as shared memory or as a file)
 - The accept mutex (depends on the implementation)
- Re-initialize modules (`ap_init_child_modules()`): Every module has been initialized by the master server before. Re-initialization is necessary if the module allocates system resources or depends on the process ID (e.g. for a database connection).
- Set up time-out handling: To avoid infinite blocking of the child server, Apache uses a time-out for the request handling. It uses alarms, a concept similar to signals. It is like setting an alarm clock to a given time and leaving the processing of the request when the "alarm bell rings". This is done using the concept of "long jump".
- Within the loop, there are two initialization steps left:
 - clear time-out: reset alarm timer
 - clear transient pool: Every memory allocation within the request-response loop concerns the transient pool. At the beginning of the loop, the pool is cleared.
- set `status := ready` in the scoreboard except after a new generation has been announced.



Accepting Connections The Preforking architecture uses an accept mutex to distribute incoming connections among multiple child servers. The *accept mutex* makes sure that only one child server process exclusively waits for a TCP request (using the system call `accept()`) — this is what a listener does. The Accept Mutex⁴ is a means of controlling access to the TCP/IP service. It is used to guarantee that, at any time, exactly one process waits for TCP/IP connection requests.

There are various implementations of the Accept Mutex for different OS configurations. Some need a special initialization phase for every child server. It works this way:

- call procedure `accept_mutex_on()`: acquire the mutex or block until available
- call procedure `accept_mutex_off()`: release the mutex

After having received a connection, the child server releases the accept mutex and processes the request — it becomes a worker and lets the next process wait for a request. This is usually called the Leader–Follower pattern: The listener is the leader, the idle workers are the followers (see figure 4.5). As Apache uses operating system dependend techniques for the mutex, it is possible depending on the operating system that all currently blocked child servers are woken when one child servers returns the mutex after receiving a connection. If so, excessive scheduling caused unnecessarily as only one of the woken child servers will get the mutex, the others will be blocked and therefore return to sleep. That is a problem which is addressed by the Leader MPM where followers are organized in a way such that only one of them is woken when the accept mutex is returned.

Once a connection is received by a child server, the scope of responsibility of the multitasking architecture ends. The child server calls the request handling routine which is equally used by any multitasking architecture.

Accept Mutex vs. `select()` In an `inetd` server (see section 4.3.1), there is only one process waiting for a TCP/IP connection request. Within the Apache HTTP server, there are possibly hundreds of idle child servers concurrently waiting for a connection request on more than one server port. This can cause severe problems on some operating systems.

Example: Apache has been configured to listen to the ports 80, 1080 and 8080. 10 Idle child server processes wait for incoming TCP/IP connection requests using the blocking system call `select()` (they are inactive until the status of one of the ports changes). Now a connection request for port 1080 comes in. 10 child server processes wake up, check the port that caused them to wake up and try to establish the connection with `accept()`. The first is successful and processes the request, while 9 child servers keep waiting for a connection at port 1080 and none at port 80 and 8080! (This worst–case scenario is only true for blocking⁵ `accept()`)

⁴A mutex is a semaphore used to enforce mutual exclusion for the access to a resource. A semaphore is a means for inter–process communication (IPC): A process can increment or decrement the value of a semaphore. The process is suspended (blocked) if it tries to decrement and the value is zero, until another process increments the value of the semaphore. To implement a mutex with a semaphore, you have to set the maximum value to 1.

⁵“blocking” means: Any process calling `accept()` is suspended. If a connection request arrives, the first resumes its operation.

Therefore in a scenario where there are multiple child servers waiting to service multiple ports the `select()` `accept()` pair is not sufficient to achieve mutual exclusion between the multiple workers. Therefore Preforking has to use the `accept` mutex.

In general it is a bad idea to waste resources of the operating system to handle concurrency. As some operating systems can't queue the child server processes waiting for a connection request, Apache has to do it.

4.3.4 Apache Multitasking Architectures and MPMs

A multiprocessing architecture's main task is to provide a fast responding server which uses the underlying operating system efficiently. Usually each architecture has to accept a trade-off between stability and performance.

In case of a HTTP server, the multitasking architecture describes the strategy how to create and control the worker tasks and how they get a request to process.

The first choice concerns the tasks: Depending on the platform, the server can use processes or threads or both to implement the tasks. Processes have a larger context (for example the process' memory) that affects the time needed to switch between processes. Threads are more lightweight because they share most of the context, unfortunately bad code can corrupt other thread's memory or more worse crash all threads of the process.

The next aspect affects the way how tasks communicate (Inter Task Communication). In general, this can be done by shared memory, signals or events, semaphores or mutex and pipes and sockets.

As all MPMs use a Task Pool strategy (idle worker tasks remain suspended until a request comes in which can immediately be processed by an idle worker task), there must be a means to suspend all idle worker tasks and wake up one whenever a request occurs. For this, an operating system mechanism like a conditional variable or a semaphore must be used to implement a mutex. The tasks are suspended when calling a blocking procedure to get the mutex.

The server sockets are a limited resource, therefore there can only be one listener per socket or one listener at all. Either there are dedicated listener tasks that have to use a job queue to hand over request data to the worker tasks, or all server tasks play both roles: One idle worker becomes the listener, receives a request and becomes a worker processing the request.

Finally a task can control the task pool by adjusting the number of idle worker tasks within a given limit.

Apache includes a variety of multitasking architectures. Originally Apache supported different architectures only to support different operating systems. Apache 1.3 had two major architectures which had to be defined at compile time using environment variables that the precompiler used to execute macros which in turn selected the correspondig code for the operating system used:

- The Preforking Architecture for Unix
- The Job Queue Architecture for Windows



Multiprocessing implementation was completely changed in Apache 2.0 by introducing *Multi Processing Modules* (MPM). These Modules are exchangeable and completely encapsulate the multiprocessing architecture. As any other module, an MPM's module structure is alike the standard module structure and includes a command table. MPMs cannot be dynamically included and therefore have to be chosen when compiling the server from the sources. Due to the nature of a multitasking architecture model, only one MPM can be included in one server at one time.

An MPM's responsibility is located within the main loops of Apache. The main server will do all initialization and configuration processing before calling the method `ap_mpm_run()` in `main()` to hand over to the MPM.

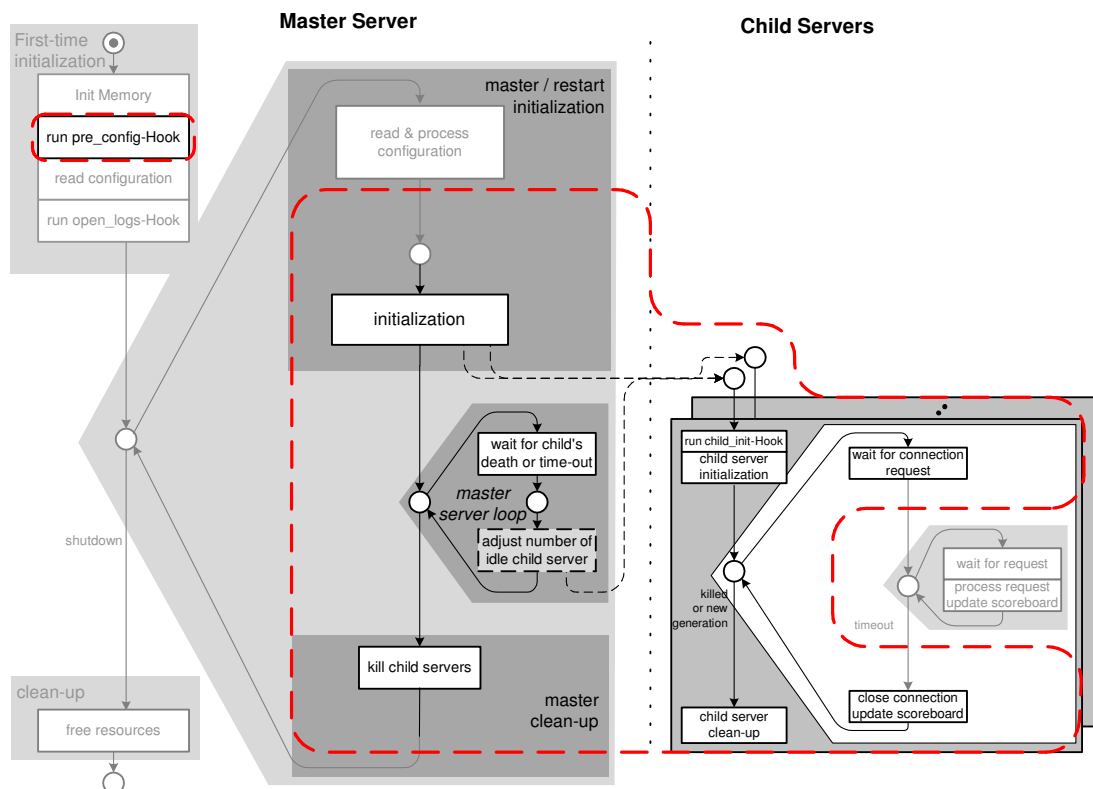


Figure 4.10: Responsibility of an Apache 2.0 MPM

It is the MPM's responsibility to take care of starting threads and/or processes as needed. The MPM will also be responsible for listening on the sockets for incoming requests. When requests arrive, the MPM will distribute them among the created threads and/or processes. These will then run the standard Apache request handling procedures. When restarting or shutting down, the MPM will hand back to the main server. Therefore all server functionality is still the same for any MPM, but the multiprocessing model is exchangeable. Figure 4.10 shows the responsibility of an Apache MPM in the overall behavior of Apache (see also figure 4.7). The dotted line marks the actions for which the MPM takes responsibility.

Version 2.0 currently includes the following MPMs:

- Preforking and Netware — MPMs that resemble the functionality of the Preforking architecture of Apache 1.3

- WinNT — Apache 1.3's Win32 version was similar to this, however the WinNT MPM is enhanced by the IOCP operating system concept
- Worker — A new MPM that makes use of both processes and threads and performs better than Preforking
- Leader and PerChild — Two MPM's still in an experimental state an alternative to Preforking and Worker on Linux based Systems

4.3.5 Win32/WinNT MPM

The Windows multitasking architecture has some significant differences compared to the preforking architecture. It uses threads instead of processes and the number of child servers is not variable but fixed. Threads are used because threads are a lot more performant than windows processes. There are two Windows processes in this multitasking architecture: The *worker process* (child server process) contains the threads which handle the HTTP requests, while the *supervisor process* (master process) makes sure that the worker process works correctly. For connection distribution to the workers a *job queue* is used. Additionally the Apache 2.0 MPM version of this multitasking architecture uses a windows NT operating concept called I/O Completion Port instead of a job queue when used on Windows NT platforms. The version 1.3 of this architecture as well as the Windows32 version of the Apache 2.0 MPM use a single listener thread to accept connections. The WindowsNT variant of the MPM uses one listener per port that Apache has to listen on.

Figure 4.11 shows the system structure of the server using the WinNT MPM: The Master Server Process creates the Child Server Process and then monitors the Child Server in case the process dies.

Initialization & Restart Loop

The initialization procedure of the Win32 multitasking architecture closely resembles the one described for the Preforking architecture. All initialization is similar to the point where the Preforking MPM is called or the Apache 1.3 architecture starts to create child server processes.

Both, the 1.3 and the 2.0 version use the master server process as the supervisor process. That in turn creates the second process that contains all the worker threads. When started the worker process only contains a single master thread which then spawns the fixed number of worker threads. These correspond to the child server processes within the Preforking architecture. The Windows multitasking version uses a fixed number of threads since idle threads impose almost no performance issue. Therefore as many threads as are desirable for optimum performance are started right away and the server can be used to its maximum capability, without the overhead and delay of spawning new processes dynamically.

Both Windows multitasking architectures only support graceful restart or shutdown.

Inter-Process/Thread Communication (events and Exit Flag)

A Variety of inter-process communication mechanisms is used in this multitasking architecture. As Windows does not support signals, the native Windows concept of *events* is used

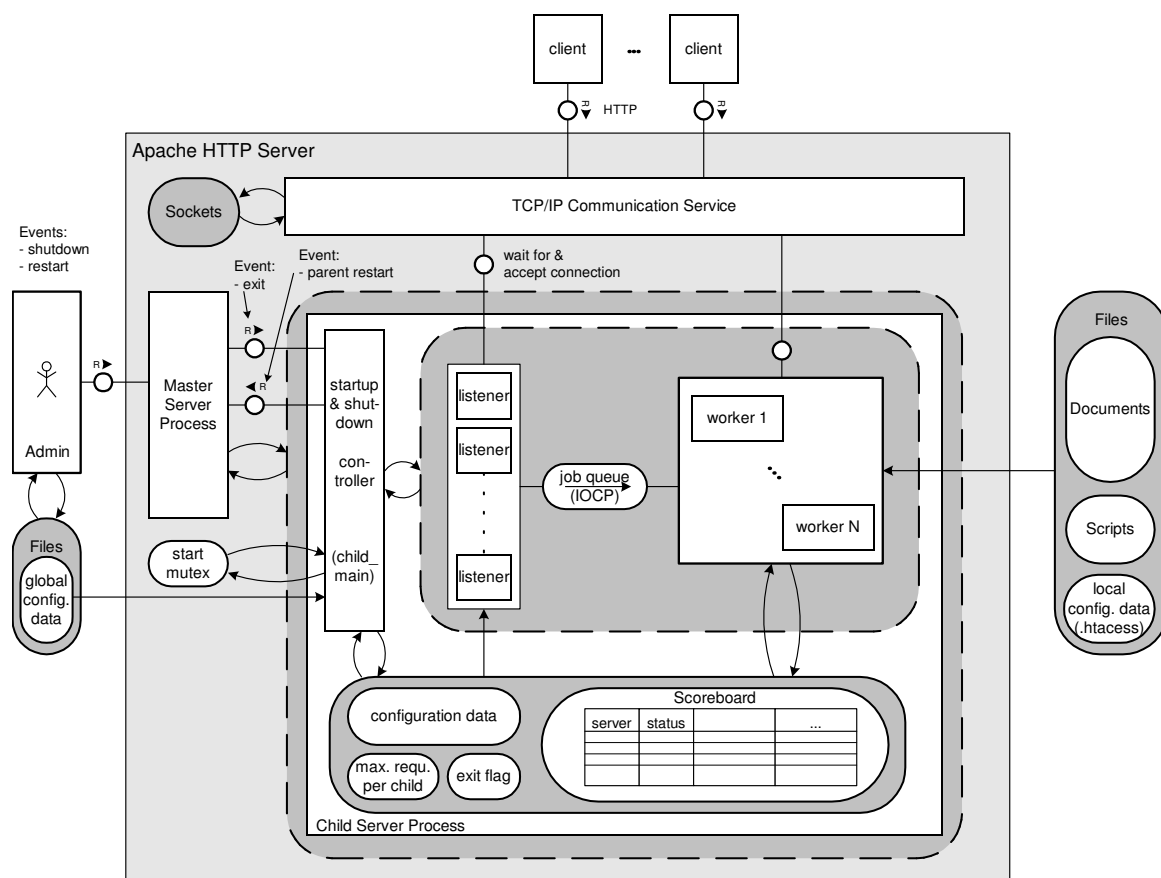


Figure 4.11: The Apache 2.0 WinNT MPM

for communication between the supervisor or master server process and the worker process. Here events are used for signaling:

- The master server process can signal the worker process that a shutdown or graceful restart is in progress.
- On the one hand the worker process can signal the master server process that it needs to be restarted or that a serious error occurred that requires to shutdown the server.

The worker process itself uses various means for communication with the listener(s) and the worker threads. When a shutdown event occurs the master thread puts "die" – jobs into the job queue or the IOCP used. Thus idle and sleeping worker threads are woken and exit, while worker threads that are busy handling a request can complete the request and quit later. Additionally it sets various exit flags that can be read by the listener(s) as well as the worker threads.

However job queue or IOCP respectively are also used by the listeners to communicate arriving connections to the worker threads for request handling.

Master Server (Supervisor) Process

The master server process is called the supervisor process when entering the restart loop. It contains only one thread and is used to monitor the second process called the worker process to be able to restart it in case it crashes. The user can communicate with this specific process using the control manager which can be found on any windows platform. The control manager then sends an event to the server which signals a restart or shutdown. Additionally the apache server supplies command line options that can be used for signaling.

Child Process: Listeners and the Job Queue

The worker process contains three kinds of threads: One master thread, a fixed number of worker threads and one or multiple listeners. The master starts one or multiple listeners which accept the connection requests and put the connection data into a job queue (like a gatekeeper). The worker threads fetch the connection data from the queue and then read and handle the request by calling the core's request handling routine which is used by all multitasking architectures. The communication between the master and the worker threads is also accomplished via the job queue. However the only communication necessary between master and worker thread is to signal a worker to exit. If the master thread wants to decrease the number of worker threads due to a pending shutdown or restart, it puts "die"–jobs into the queue.

Instead of a selfmade job queue, the MPM of Version 2.0 uses the IOCP on Windows NT platforms. The advantage of the I/O Completion Port is that it enables the server to specify an upper limit of active threads. All worker threads registering with the IOCP are put to sleep as if registering with a job queue. When any of the events the IOCP is responsible for occurs one worker thread is woken to handle the event (in Apache that can only be a new incoming connection request). If however the limit of active threads is exceeded, no threads are woken to handle new requests until another thread blocks on a synchronous call or reregisters with the IOCP. That technique is used to prevent excessive context switching and paging due to large numbers of active threads.



Child Process: Worker Threads

Worker Threads are kept pretty simple in this architecture model. As they can share any memory with their parent process (the worker process) they do not need to initialize a lot of memory. All they maintain is a counter of requests that the thread processed. The MPM version also keeps a variable containing the current operating system version so that either the job queue or the IOCP is chosen when trying to get a connection. Therefore the initialization is very short.

After initialization the worker registers with the IOCP or the job queue to retrieve a connection which it can handle. After receiving the connection it calls the core's request processing routine. It continues to do that until it is given a “die”-job from the queue, which would cause it to exit.

4.3.6 Worker MPM

The Worker MPM is a Multiprocessing Model for the Linux/Unix Operating System Platform. In contrast to the Preforking and WinNT Model, this MPM uses a combination of a multiprocessing and a multithreading model: It uses a variable number of processes, which include a fixed number of threads (see figure 4.12). The preforking model on process level is extended by a job queue model on thread level.

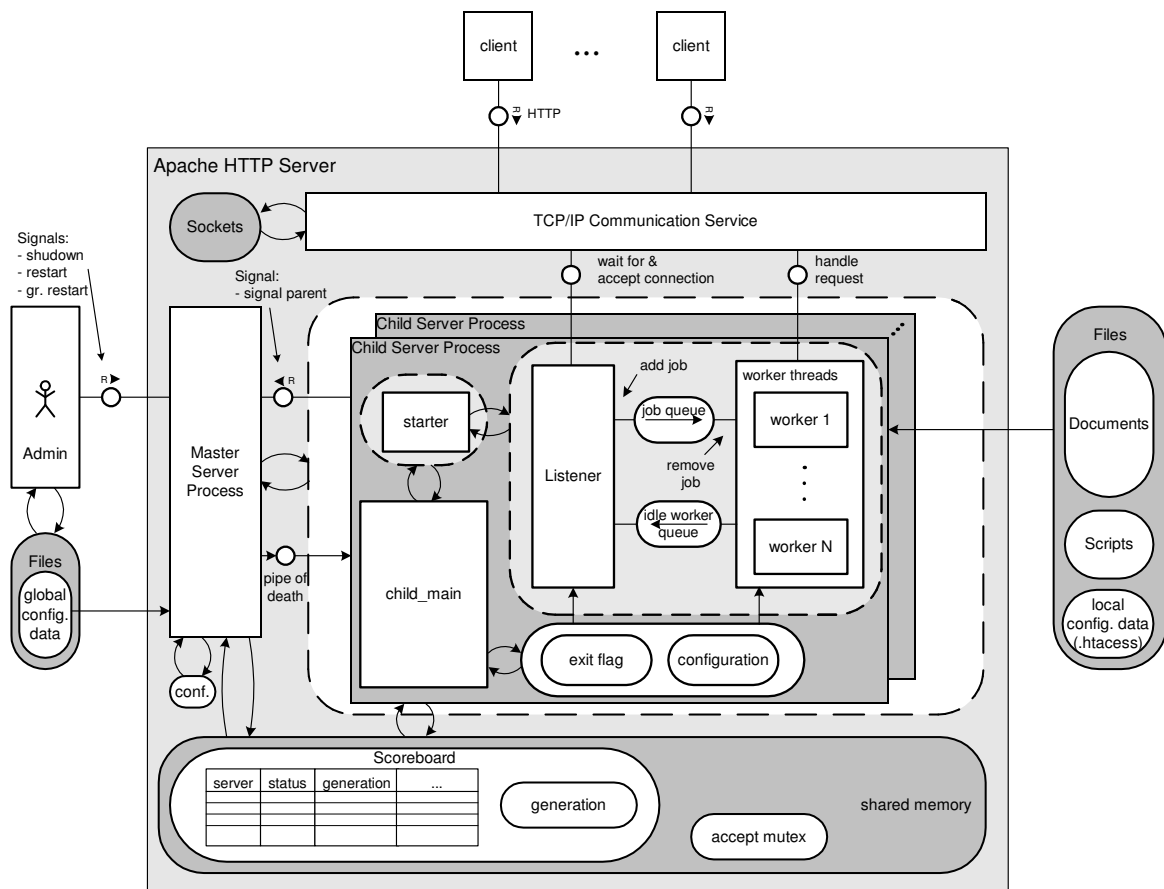


Figure 4.12: The Apache 2.0 Worker MPM

Still the master server process adjusts the number of idle processes within a given range based on server load and the `max_child`, `max_idle` and `min_idle` configuration directive. Each child process incorporates a listener thread, which listens on all ports in use by the server. Multiple processes and therefore multiple listener threads are mutually excluded using the accept mutex like in the Preforking Model.

Child Server / Worker

Initialization of a child server is a more complex task in this case, as a child server is a more complex structure. First the master server creates the child process, which in turn starts a so called starter thread that has to set up all worker threads and a single listener thread. This behavior is reflected in figure 4.12.

Within each child process, the communication between the listener and all worker threads is organized with two queues, the *job queue* and the *idle queue*. A listener thread will only apply for the accept mutex if it finds a token in the idle queue indicating that at least one idle worker thread is waiting to process a request. If the listener gets the mutex, it waits for a new request and puts a job item into the queue after releasing the accept mutex. Thus it is ensured that a incoming request can be served by a worker immediately.

After completing a request or a connection with multiple requests (see section 2.3.4 for details) the worker thread registers as idle by putting a token into the idle queue and returns to wait for a new item in the worker queue.

Advantages of this approach are that it combines the stable concept of multiprocessing with the increased performance of a multithreading concept. In case of a crash, only the process that crashed is affected. In multithreading a crashing thread can affect all threads belonging to the same parent process. Still threads are a lot more lightweight and therefore cause less performance overhead during start-up and consume less memory while running.

4.3.7 Others MPMs

The MPMs mentioned so far are the MPMs used most often. Additionally there are other MPMs available. However most of these mainly serve an experimental purpose and are seldom used in productive environments.

Leader MPM

This MPM uses the preforking (Leader-Follower, see also figure 4.5 and the pattern description in [5]) model on both process and thread level using a sophisticated mechanism for the followers queue:

Each Child Process has a fixed number of threads like in Worker MPM. However, threads are not distinguished into worker and listener threads. Idle workers are put onto a stack. The topmost worker is made listener and will upon receiving a connection immediately become a worker to handle the connection itself. The worker on the stack below him will become the new listener and handle the next request. Upon finishing a request the worker will return to the top of the stack.



This approach addresses two performance issues. First there is no delay due to handing a connection to a different task using a job queue, since each thread simply handles the connection it accepts. Secondly since follower threads are organized in a stack, only one thread is woken when the listener position becomes available. The overhead that is caused when all threads are woken to compete for the mutex is avoided.

A thread returning to the stack is put on top. Therefore it is most likely that a thread on top will handle more requests than a thread at the bottom. Considering the paging techniques for virtual memory that most operating systems use, paging is reduced as more often used threads do more work and thus are less likely to be swapped to the hard disk.

Per-Child MPM

Based on the Worker MPM, this experimental MPM uses a fixed number of processes, which in turn have a variable number of threads. This MPM also uses the preforking model on both process and thread level. The advantage of this approach is that no new processes have to be started or killed for load balancing.

An advantage of this MPM: Each process can have a separate UserID, which in turn is associated with different file access and program execution rights. This is used to implement virtual hosts with different rights for different users. Here each virtual host can have its own process, which is equipped with the rights for the corresponding owner, and still the server is able to react to a changing server load by creating or destroying worker threads.

4.4 The Request–Response Loop

4.4.1 Overview

The Request–Response Loop is the heart of the HTTP Server. Every Apache child server processes this loop until it dies either because it was asked to exit by the master server or because it realized that its generation is obsolete.

Figure 4.13 shows the request–response loop and the keep–alive loop. To be exact, the request–response loop deals with waiting for and accepting connection requests while the keep–alive loop deals with receiving and responding to HTTP requests on that connection.

4.4.2 Waiting for connection requests

Depending on the multitasking architecture used, either each idle worker tries to become listener, or it waits for a job in a job queue. In both cases it will be suspended until the mutex or the job queue indicates either that it will be the next listener or that a new job is in the queue.

The transitions in the rectangle “wait for TCP request” in figure 4.13 show the Leader–Follower model of the Preforking MPM: The child server task tries to get the *accept mutex* to become listener and will be suspended again until a TCP connection request comes in. It accepts the connection and releases the accept mutex. (see also `child_main()` in `prefork.c`).

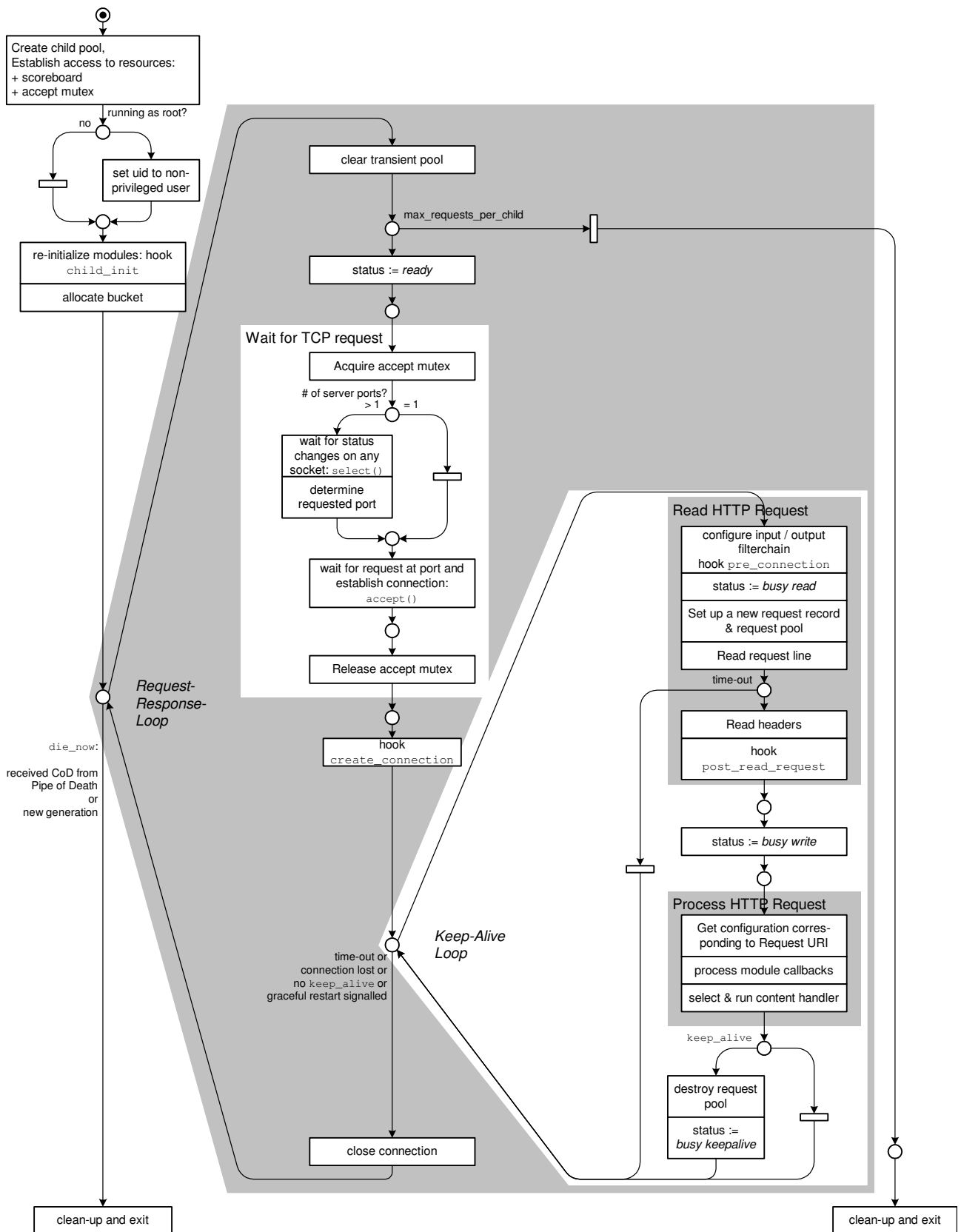


Figure 4.13: The Request-Response Loop



After a child server received a connection request, it leaves the scope of the MPM and triggers the hooks `pre_connection` and `process_connection`. The module `http_core.c` registers the handler `ap_process_http_connection()` for the latter hook which reads and processes the request.

4.4.3 Waiting for and reading HTTP requests

An HTTP client, for example a browser, re-uses an existing TCP connection for a sequence of requests. An HTML document with 5 images results in a sequence of 6 HTTP requests that can use the same TCP connection. The TCP connection is closed after a time-out period (usually 15 seconds). As the HTTP header used to control the connection had the value “keep-alive”, the loop carries this name.

The keep-alive loop for reading and processing HTTP requests is specific for HTTP. Therefore in Apache 2, the module `http_core.c` registers the handler `ap_process_http_connection()` which includes the keep-alive loop. Similar to the transient pool, the request pool is cleared with every run of the keep-alive loop.

The child server reads the request header (the request body will be treated by the corresponding content handler) with the procedure `ap_read_request()` in `protocol.c`. It stores the result of the parsing in the data structure `request_rec`. There are many errors that can occur in this phase⁶. Note that only the header of the HTTP request is read at this time!

4.4.4 Process HTTP Request

After the HTTP header has been read, the child server status changes to “*busy_write*”. Now it’s time to respond to the request.

Figure 4.14 shows the details of the request processing in Apache 2.0. Request processing in Apache 1.3 is almost the same. The only major exception is that only a single content handler can be used in apache, but multiple modules can take part in forming the response in Apache 2.0 as the filter concept is used.

The procedure `ap_process_request()` in `http_request.c` calls `process_request_internal()` in `request.c`. What happens in this procedure is shown in figure 4.14 which is similar to figure 3.9 in section 3.3 on page 44, but provides technical details and explanations:

- First the URI is modified (`ap_unescape_URL()`, `ap_getparents()`): Apache replaces escape character sequences like “%20” and path ‘noise’ like “./xxx” or “xxx/./”.
- Then Apache retrieves the configuration for the Request URI: `location_walk()`. This happens before a module translates the URI because it can influence the way the URI is translated. Detailed information about Apache’s configuration management can be found in section 4.5.

⁶If you want to check this, establish a TCP/IP connection with `telnet (telnet hostname port)` and type in a correct HTTP request according to HTTP/1.1.

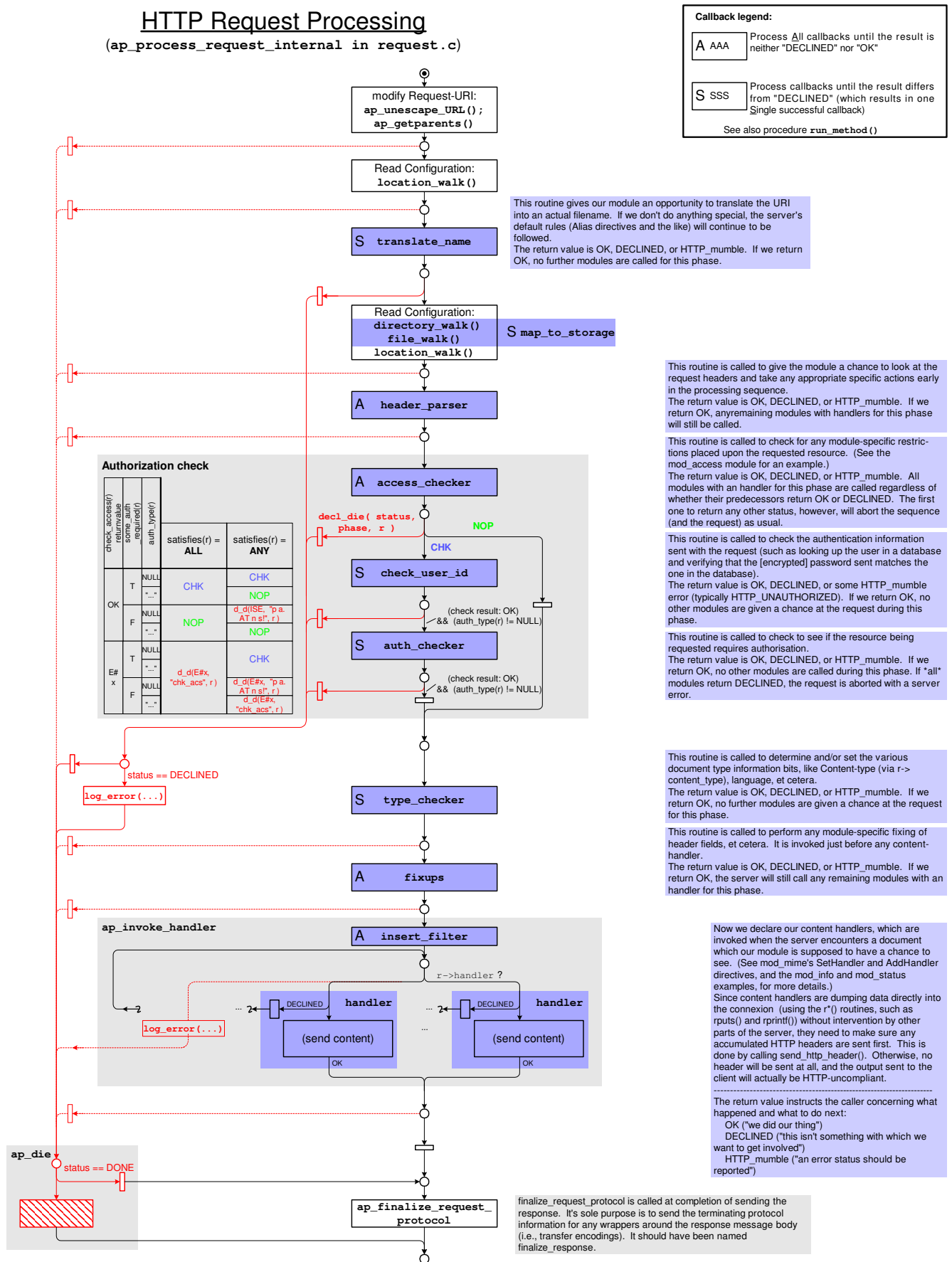


Figure 4.14: Processing of a request



- `ap_translate_name()`: Some module handler must translate the request URI into a local resource name, usually a path in the file system.
- Again it gets the pieces of configuration information for the Request URI with `location_walk()` (the URI can differ from the request URI after being processed by `ap_translate_name()`). The core handler for the hook `map_to_storage`, `core_map_to_storage()` calls `ap_directory_walk()` and `ap_file_walk()` which collect and merge configuration information for the path and the file name of the requested resource. The result is the configuration that will be given to all module handlers that process this request.
 (“walk”: Apache traverses the configuration information of every section of the path or URI from the root to the leaf and merges them. The `.htaccess` files are read by `directory_walk()` in every directory and by `file_walk()` in the leaf directory.)
- `header_parser`: Every module has the opportunity to process the header (to read cookies for example).

Authorization check There are two independent authorization checks:

1. Access check based on the IP address of the client computer
2. Authorization check based on the identity of the client user (to get the identity, an authentication check is necessary)

The administrator can configure for each resource:

- users, groups, IP addresses and network addresses
- the rules for the authorization check (allow/deny IP or users, either both IP and Identity check must be successful or only one of both)

The complex behavior of the authorization check could not be illustrated completely in figure 4.14. Use the matrix on the left-hand side to map the program statements to the operations.

- `access_checker`: IP-based authorization check
- `ap_check_user_id`: authentication check
- `auth_checker`: authorization check based on the user’s identity
- `type_checker`: get the MIME type of the requested resource. This is important for selecting the corresponding content handler.
- `fixups`: Any module can write data into the response header (to set a cookie, for example).
- `insert_filter`: Any module can register output filters.

- **handler:** The module registered for the MIME type of the resource offers one or more content handlers to create the response data (header and body). Some handlers, e.g. the CGI module content handler, read the body of the request. The content handler sends the response body through the output filter chain.
- `ap_finalize_request_protocol()`: This method should have been named “finalize response”. Its sole purpose is to send the terminating protocol information for any wrappers around the response message body (i.e. transfer encodings).

The error handling shown on the left side is ‘misused’ if Apache wants to skip further request processing. The procedure `ap_die()` checks the status and sends an error message only if an error occurred. This “misuse” happens for example if `ap_translate_name` is successful (it returns “DONE”)!

For more information on filters, check section 3.3.4.

4.5 The Configuration Processor

4.5.1 Where and when Apache reads configuration

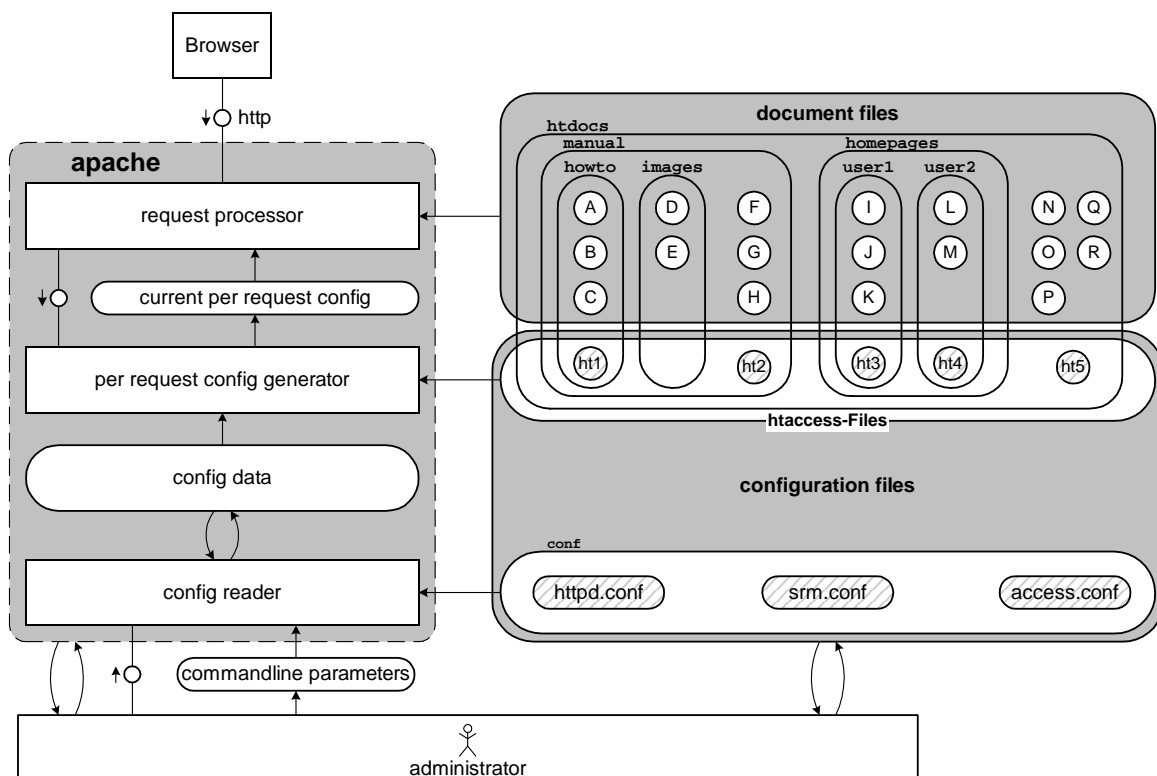


Figure 4.15: Configuration processing components of Apache

After Apache has been built, the administrator can configure it at start-up via command line parameters and global configuration files. Local configuration files (usually named `.htaccess`) are processed during each request and can be modified by web authors.



Figure 4.15 shows the system structure of Apache focusing on the configuration processor. At the bottom, we see the administrator modifying global configuration files like `httpd.conf`, `srn.conf`, `access.conf` and local configuration files `ht1` to `ht5` (`.htaccess`). The administrator starts Apache passing command line parameters. The config reader then reads and processes the global configuration files and the command line parameters and stores the result in the config data storage, which holds the internal configuration data structures.

For each request sent from a browser, the request processor advises the 'per request config generator' to generate a per-request config valid for this request. The per request config generator has to process the `.htaccess` files it finds in the resource's path and merges it with the config data. The request processor now knows how to map the request URI to a resource and can decide if the browser is authorized to get this resource.

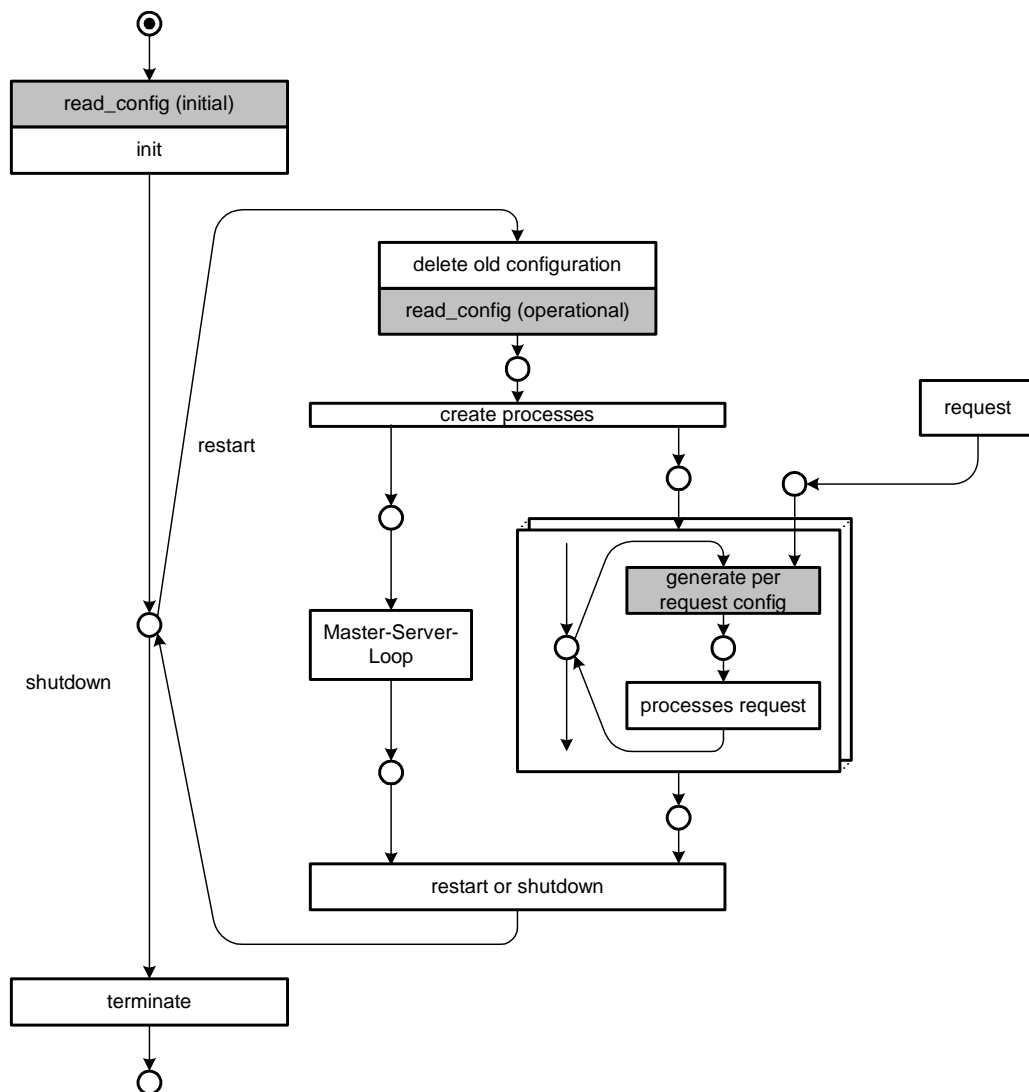


Figure 4.16: Points in time when Apache reads configuration

Figure 4.16 shows the situations when Apache processes configuration — the diagram is an excerpt of figure 4.7. After the master server has read the per-server configuration, it enters the 'Restart Loop'. It deletes the old configuration and processes the main configuration file again as it does it on every restart.

In the 'Request-Response Loop', a child server generates the per-request configuration for each request. As the configuration data concerning the request is tied to the per-request data structure `request_rec`, it is deleted after the request has been processed.

In the next parts, we will first take a look on the data structures that are generated when processing the global configuration files. After that we take a look at the source code responsible for doing this. Then we describe the processing of configuration data on requests.

4.5.2 Internal data structures

We will discuss the internal configuration data structures using figure 4.17 which gives an overview on the data structures in an Entity Relationship Diagram, and figure 4.18 which shows an example structure with the module `mod_so`.

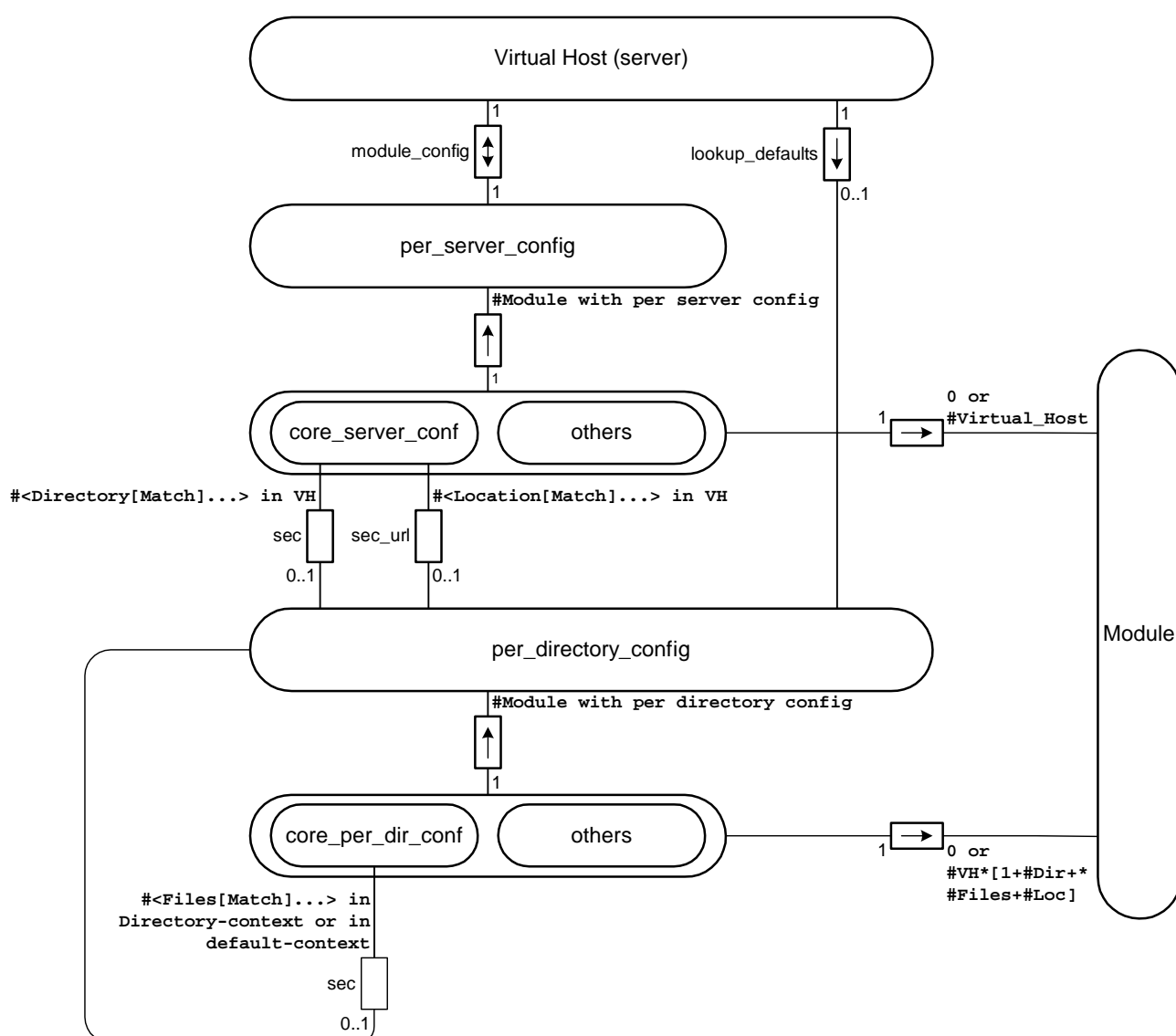


Figure 4.17: Apache data structures concerning configuration

Each virtual host has exactly one `per_server_config` containing per-server information and one `per_directory_config` containing defaults for the per-directory configuration

(lookup_defaults). `per_server_config` and `per_directory_config` are implemented as pointers to arrays of pointers.

For each module, the `per_server_config` array contains an entry. The entry for the core module (`core_server_conf`) has again a `per_directory_config` for each `Directory` and `Location` directive found in the configuration files.

Each `per_directory_config` array again contains an entry for each module. The entry for the core module (`core_per_dir_conf`) has again a `per_directory_config` for each `Files` directive found in the configuration files.

Figure 4.18 shows the data structure generated by the core module for each virtual host. `server_conf` points to the actual virtual server, the appropriate one is found by traversing the list of hosts via the 'next'-pointers in the `server_recs`. These also contain other per-server configuration like `server_admin`, `server_hostname` and pointers to the `module_config` and `lookup_defaults` data structures.

Both point to an array whose size is the total number of modules plus the number of dynamically loadable modules, i.e one entry for each possibly available module. Each field of the array points to a data structure defined by the accordant module (via the per-server config handlers).

In figure 4.18 the data structures of the core module and `mod_so` are shown, the one for `mod_so` just being the list of loaded modules. The focus of the diagram is on the data structure of the core module, as it is essential for the configuration of Apache, which is composed of `document_root`, `access_name`, `sec_url` and `sec`.

`sec` points to an array consisting of pointers, one pointer for each directory section found in the configuration files and ordered by the number of slashes in the path, shortest to longest. Each of them in turn points to an array, which contains an entry for each module. Hence, the `command_handlers` of each module have the possibility to create data structures and to make entries in the corresponding section of each directory (through the per-directory config handlers). The data structure for the core module of each directory section again contains a `sec` pointer to a similar data structure as before, this time for the files sections.

Beneath `sec` the module data structure of the core module contains a pointer called `sec_url`. This one also points to a similar data structure as the other `sec` pointers mentioned before, but for the location sections in the configuration files.

So, every module gets the opportunity to build its own data structure for each directory, files and location directive. The per-directory config handlers are also responsible for the entries corresponding to the file and location sections.

The data structure which `lookup_defaults` points to is again similar to the already explained ones. However, its `sec` pointer just points to an array for file sections. This is because it contains data of files sections, which are not in a specific directory context.

Additionally, you can see a part of the module structures at the bottom of the diagram, showing the `command_rec` containing some of the directives of the core and `mod_so` modules.

4.5.3 Processing global configuration data at start-up

On start-up, Apache eventually processes the main configuration file twice (see figure 4.16). The first pass is necessary, for example, for syntax checking of the global configuration file



(command line parameter `-t`) or if Apache runs in `inetd` mode. The second pass is necessary, as Apache needs to get the actual configuration on every restart.

Apache calls the function `ap_read_config()` for processing the configuration when starting or restarting. The function is called for the first time in `main()` and afterwards in the 'Restart Loop'.

Processing global configuration data

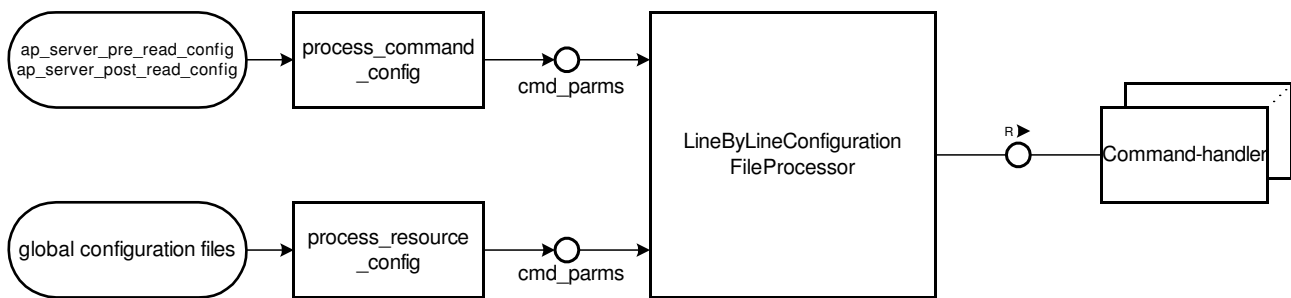


Figure 4.19: Structure of the configuration processor (focusing on the data flow)

Figure 4.19 shows the data flow in the structure of the global configuration processor:

The agent `process_command_config` is responsible for reading command line parameters from the storages `ap_server_pre_read_config` and `ap_server_post_read_config`, while the agent `process_resource_config` reads the global configuration files. Both agents pass their data to the Line-by-line configuration file processor (`ap_srm_command_loop`). This is the heart of the configuration processor and it schedules the processing of a directive to the corresponding command handlers in the modules.

Figure 4.20 shows the layering of function calls regarding configuration⁷ (Note: Only the most important procedures are covered.):

`ap_read_config()` calls the procedures `process_command_config()` and `ap_process_resource_config()`.

`process_command_config()` processes directives that are passed to Apache at the command line (command line options `-c` or `-C`). The arguments are stored in the arrays `ap_server_pre_read_config` and `ap_server_post_read_config` when reading command line options in `main()`, depending on if they should be processed before or after the main configuration file. These arrays are now handled like configuration files and are passed to the function `ap_build_config()` (`ap_srm_command_loop()` in Apache 1.3) in a `cmd_parms` data structure, which contains additional information like the affected server, memory pointers (pools) and the override information (see also figure 4.20).

`ap_process_resource_config()` actually processes the main configuration file. Apache has the ability to process a directory structure of configuration files, in case a directory name instead of a filename was given for the main configuration file. The function calls itself recursively for each subdirectory and so processes the directory structure. For each file that has

⁷Layer diagram of function calls: A line crossing another line horizontally in a circle means the box where the line starts contains the name of the calling procedure, whereas the box with the line that crosses vertically contains the name of the called one.

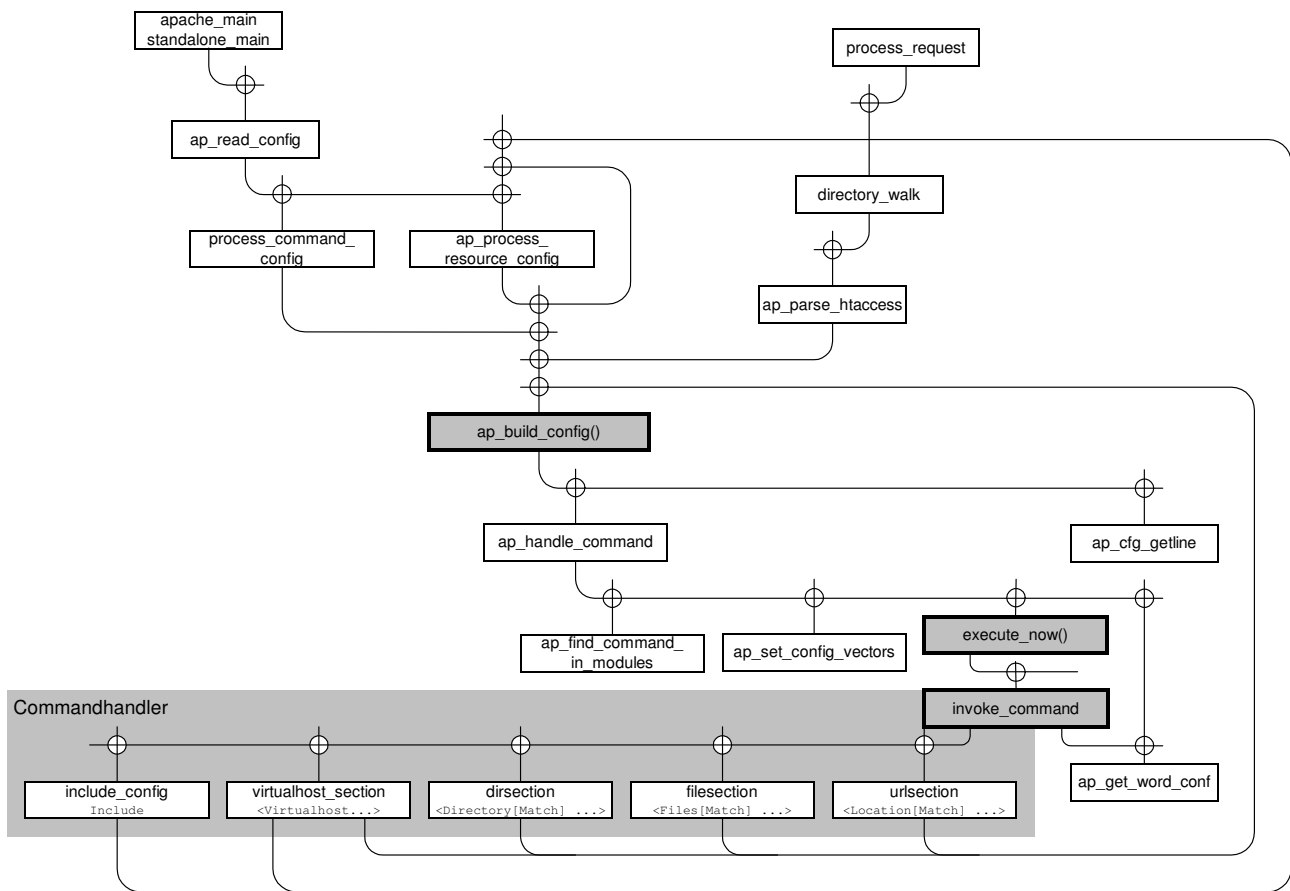


Figure 4.20: Reading configuration data: Layering of function calls



been found at the recursion endpoint, a `cmd_parms` structure containing a handle to the configuration file is initialized and passed to `ap_build_config()` (`ap_srm_command_loop()` in Apache 1.3).

Processing a Directive

`ap_build_config()` (`ap_srm_command_loop()` in Apache 1.3) processes the directives in the configuration file on a line by line basis. To accomplish that, it uses the function `ap_cfg_get_line()` (`ap_cfg_get_line()` in Apache 1.3) which returns one line of the configuration file it has parsed, removing leading and trailing white space, deleting backslashes for line continuation and so on.

Afterwards, this line is passed to `ap_build_config_sub()` (`ap_handle_command()` in Apache 1.3), which just returns doing nothing if the line is empty or a commentary line. Otherwise it uses `ap_find_command_in_modules()` (`ap_find_command_in_modules()` respectively) to find the first module in the list of modules that could handle the command whose name it has extracted from the line before. The returned `command_rec` structure (see section 3.3 on modules for additional information on the `command_rec` structure) is passed to the `procedureexecute_now()` which in turn executes `invoke_cmd()` (only `invoke_cmd()` in Apache 1.3). If `execute_now()` (or `invoke_cmd()`) returns a `DECLINED`, `ap_find_command_in_modules()` is called again to get the `command_rec` of the next module that could handle the command.

`invoke_cmd()` is the procedure that actually invokes the function in the module via a function-pointer. Depending on the information in the `command_rec`, the adequate number of arguments is extracted (`ap_get_word_conf()` in 2.0 or `ap_get_word_conf()` in 1.3) and passed to the called function, as are the `cmd_parms` structure and the `module_config`. The `cmd_parms` structure contains the information where the handler can write its configuration information.

Processing Directory, Files and Location sections

Figure 4.20 also shows some of the command handlers of the core module (Links in Brackets show the Apache 1.3 version):

`dirsection` (`dirsection`), `filesection` (`filesection`) and `urlsection` (`urlsection`) are the corresponding functions to the `<Directory>`, `<Files>` and `<Location>` directives. Again they use `ap_build_config()` (`ap_srm_command_loop()`) to handle the directives inside a nested section.

As an example, we take a look at how Apache processes a `<Directory>` directive by invoking the command handler `dirsection()`. This can be seen in figure 4.21. Now `dirsection()` calls for `ap_build_config()` (`ap_srm_command_loop()`) to process all directives in this nested sections line by line.

If Apache detects the directive `</Directory>`, it invokes the corresponding command handler, which returns the found `</Directory>` string as an error message, so the processing of lines is stopped and `ap_build_config()` (`ap_srm_command_loop()`) returns. If it returns `NULL` it has finished processing the configuration file and has not found the corresponding end section tag. The calling `dirsection()` function returns a 'missing end section' error.

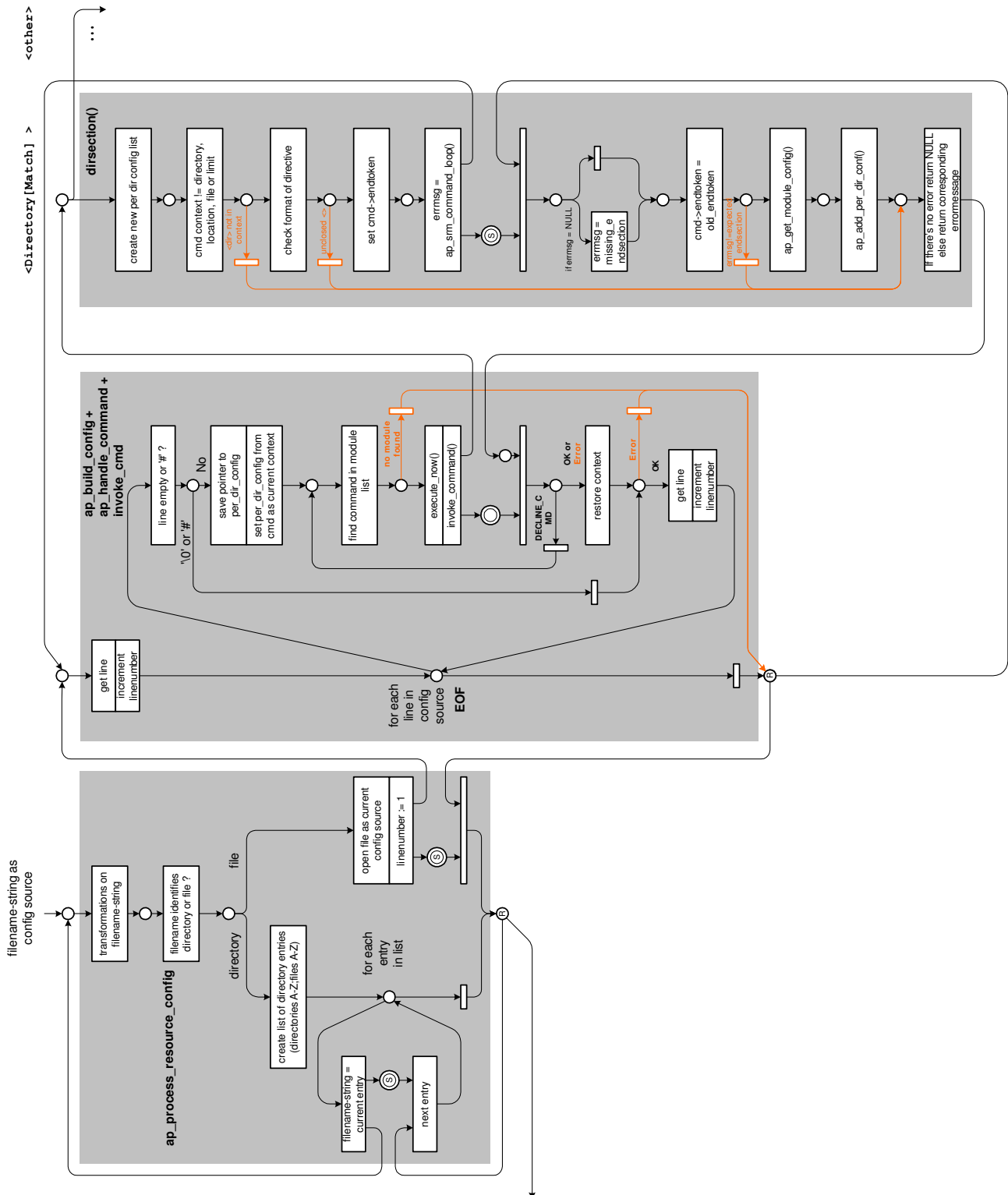


Figure 4.21: Processing configuration files with section directives



Otherwise, it adds a per-directory configuration array to the data structures of the corresponding server (virtual host). The `end_nested_section()` function knows for which section end it has to look because the name is stored in the `cmd_parms` structure which is passed to the command handlers.

The `<VirtualHost>` directive works similarly, the `Include` directive just calls again `ap_process_resource_config()` to process an additional configuration file.

4.5.4 Processing configuration data on requests

Affected data structures

Apache reads configuration data at start-up and stores it in internal data structures. Whenever a child server processes a request, it reads the internal configuration data and merges the data read from the `.htaccess` files to get the appropriate configuration for the current request.

Figure 4.22 presents the system structure of the configuration processor and its storages containing internal configuration data structures for one virtual host. These configuration data structures have been generated at start-up and are presented in detail in figures 4.17 and 4.18. Here, the `sec`, `sec_url`, `server_rec` and `lookup_defaults` structures are shown. The name of the files that are to be processed on a request is also stored in the `core_server_config` and is `.htaccess` by default. The configuration for the request is generated in the `request_rec` data structure, which is represented on the right side of the diagram and also provides other required information to the walk functions.

Invoked functions

The child server invokes `process_request_internal()` to process a request. It first retrieves the configuration for the URI out of the `sec_url` structure, by calling the `location_walk()` procedure and passing it the `request_rec` as a parameter. This is done before a module handler gets the possibility to translate the request URI (`ap_translate_name()`), because it can influence the way the URI is translated.

After translating the request URI, the child server calls the three walk procedures in the order `directory-`, `file-` and `location-walk` (see figure 4.14).

The walk procedures

The `directory_walk()` procedure clears the `per_dir_config`, so the first `location-walk` before URI translation does not influence the further processing here.

The walk procedures all work similarly. They traverse the list of entries in the corresponding data structures and check for matching `directory-`, `files-`, or `URI-sections`.

However, the `directory-walk` is somewhat more complicated and will therefore be presented in more detail:

The `directory-walk` goes down the directory hierarchy searching for directory names, which apply to the name in the `request_rec` and merges the corresponding entries in the

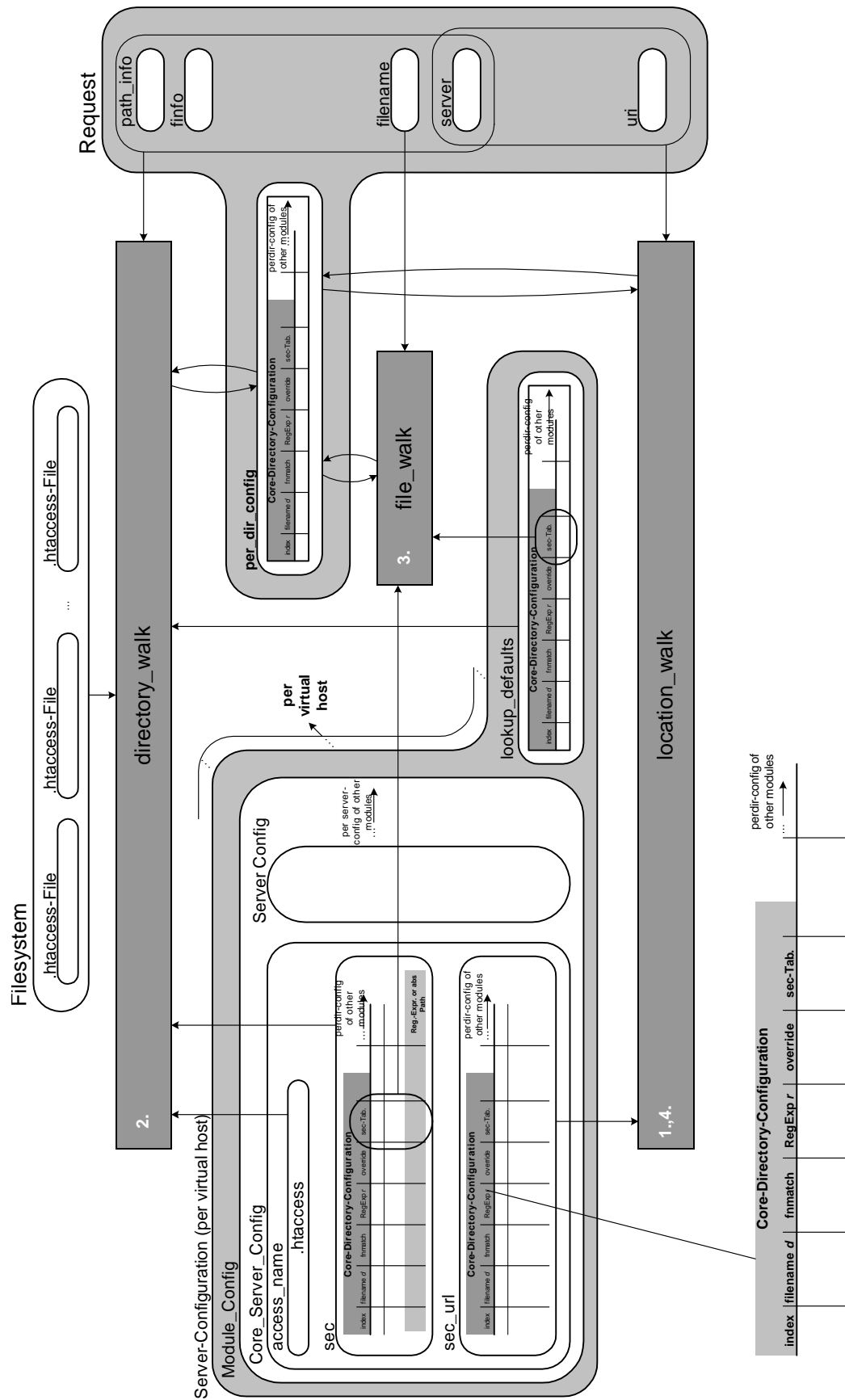


Figure 4.22: Structure of the per-request configuration processor: The walk procedures

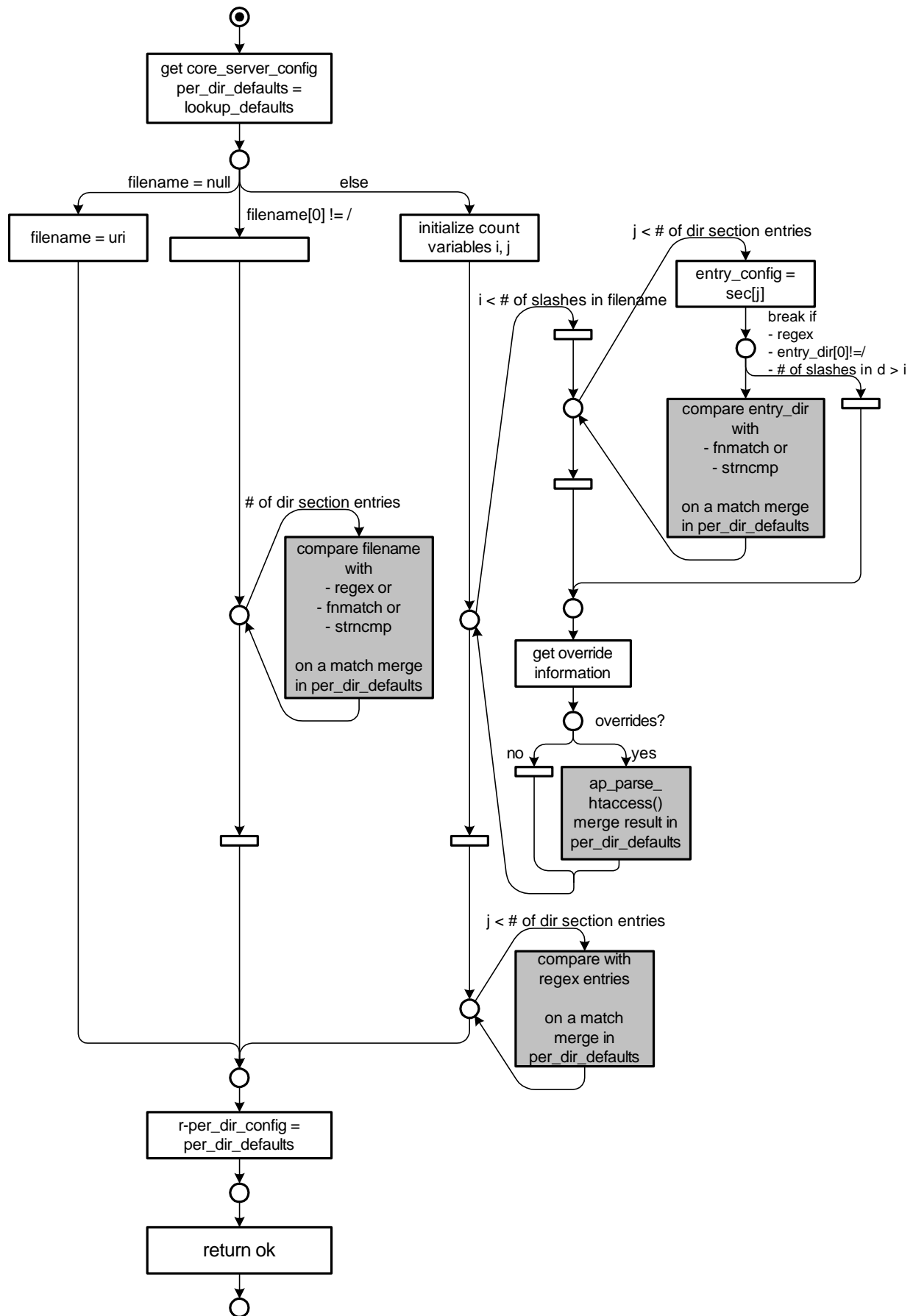


Figure 4.23: The directory walk (without error handling)

`per_dir_config` of the `request_rec` (using `ap_merge_per_dir_configs()`). Figure 4.23 shows what happens during the directory-walk. (Error handling has been left out.):

First, the `lookup_defaults` structure is assigned to `per_dir_defaults` and therefore taken as the basis for further processing. Later, all matching directory sections are merged to the `per_dir_defaults` (in a pool in the `request_rec`).

Mainly, there are three important ways the directory-walk can take, depending on the content of `filename`. Each path results in the assignment of `per_dir_defaults` to the `request_rec`, thus, enabling the file-walk to work on a data structure that contains all relevant file section entries.

1. The left path is taken if there is no `filename` given and just sets the URI as `filename`.
2. The second path is the one taken if `filename` is not starting with a `'/'`. Here, the directory-walk just loops through the directory section entries and compares `filename` to the entries.

For comparing it uses either the entry `fnmatch`, using a compare function specified in POSIX, a regular expression entry or simply a string compare with `filename`.

It loops through the array of entries and tests each section if it matches and merges it on a hit in `per_dir_defaults`.

3. The third path uses a nested loop. Here, the order of the entries is of importance (see `ap_core_reorder_directories()` in `http_core.c`). The directory sections are ordered such that the 1-component sections come first, then the 2-component, and so on, finally followed by the 'special' sections. A section is 'special' if it is a `regex` (regular expression), or if it doesn't start with a `'/'`.

The outer loop runs as long as the number of slashes in `filename` is larger than `i` (a counter which is incremented on each pass). If `i` is larger the possibly matching sections are already passed.

The nested loop actually walks through the entries, memorizing its position in `j`. If the actual entry is a regular expression or if the directory name is not starting with a `'/'`, the inner loop breaks because it has entered the 'special' sections and the outer loop is finished, too. Regular expressions are compared later on in a separate loop.

If the inner loop breaks because the number of slashes in the directory name of the entry is larger than `i`, the entry is skipped and the `.htaccess` file of the corresponding directory is included if allowed. Then the outer loop starts a new cycle. This way, all relevant `.htaccess` files are included.

If no break occurs we are in the right place. In the inner loop the directory name of the entry is compared with `fnmatch` or `strncmp`. On a match the result is merged in `per_dir_defaults`.

The override information is applied and where a `.htaccess` file has the permission to override anything, the method tries to find one.

If a `.htaccess` file has to be parsed, `ap_parse_htaccess()` is invoked. This procedure in turn calls `ap_build_config()` (see figure 4.20), which works the same way as at start-up for the main configuration files, but this time on the `per_dir_config` structure of the `request_rec`.

`file_walk()` works only on the `per_dir_config` of the `request_rec` because the structures for the file directives are already copied from the `core_server_config`'s and



`lookup_defaults`' file section tables to the `per_dir_config` of the `request_rec` by the `directory_walk`. The filename of the file to look for is provided by the `request_rec`.

`location_walk()` uses the `sec_url` and the URI provided by the `request_rec` to work on the `per_dir_config` of the request. As the other walk-functions, it loops through the entries of its corresponding data structure (`sec_url`) and merges matching entries to the `request_rec`.

4.6 Memory and resource management: Apache Pools

Apache provides an own memory and resource management known as pools. Pools are means to keep track of all resources ever used by Apache itself or any add-on module used with Apache. A pool can manage memory, sockets and processes, which are all valuable resources for a server system.

4.6.1 Why another memory management

First of all pools are used to reduce the likelihood of programming errors. In C any program can allocate memory. If a program loses all references to that memory without marking the memory as free to use for any other program, it cannot be used by anyone until the program exits. This is called a memory leak. In programs mainly used in a desktop environment this does not do much harm since programs do not run very long and when exiting, the operating system frees all memory anyway. In a server environment the server software should potentially run for a very long time. Even small memory leaks in program parts that are repeatedly run can cause a server to crash.

Sockets are similarly sensitive. If a program registers for a socket and does not free that socket once it finished using it, no other program will be able to use it. Sockets also cause other resources like memory and CPU capacity to be occupied. If a part of a server program that is run repeatedly registers sockets and does not free them, the server will inevitably crash. Many servers work multithreaded which means more than one process or thread will run to do work for the server. Usually these processes will only be terminated by the administrator, if they do not exit by design. If a program starts never-ending processes repeatedly and forgets to ask them to terminate, the server's resources will be maxed out quickly and the server will crash.

With the pool concept, a developer registers any memory, socket or process with a pool that has a predetermined lifetime. Once the pool is destroyed, all resources managed by that pool are released automatically. Only a few routines that have been tested thoroughly will then take care of freeing all resources registered for this pool. That way only a few routines have to make sure that all resources are freed. Mistakes within those are a lot easier to find and this technology takes a burden of all other developers.

Additionally the pool technology improves performance. Usually a program uses memory once and deallocates it. That causes a lot of overhead, as the system has to allocate and deallocate and map to and from virtual memory. When using a lot of small bits of memory, that can reduce performance dramatically. Also, the system usually allocates a minimum number of bytes no matter how much memory was requested. That way small bits of memory are wasted. When done too often, that can add up to an amount worth taking care of.

4.6.2 The pool structure

Built-in Pools and lifetime

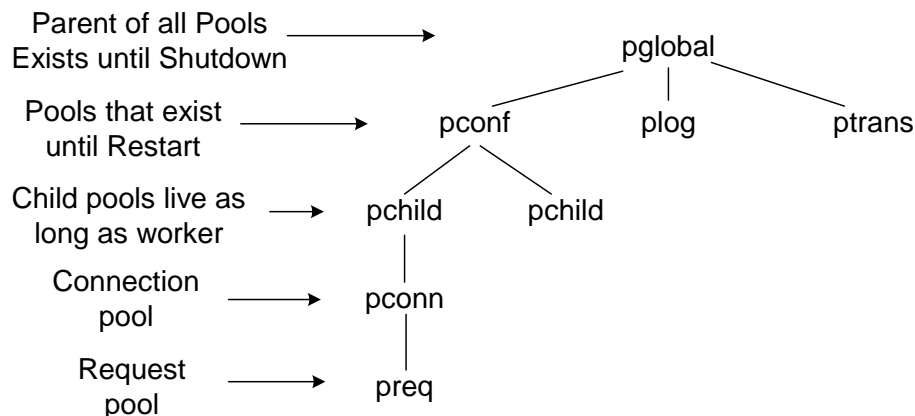


Figure 4.24: Hierarchy of built-in Apache Pools

Apache has different built-in pools that each have a different lifetime. Figure 4.24 shows the hierarchy of the pools. The pool `pglobal` exists for the entire runtime of the server. The `pchild` pool has the lifetime of a child server. The `pconn` pool is associated with each connection and the `preq` pool for each request. Usually a developer should use the pool with the minimum acceptable lifetime for the data that is to be stored to minimize resource usage.

If a developer needs a rather large amount of memory or other resources and cannot find a pool with the acceptable lifetime, a sub pool to any of the other pools can be created. The program can then use that pool like any other and can additionally destroy that pool once it is not needed any longer. If the program forgets to destroy the pool, it will automatically be destroyed once the Apache core destroys the parent pool. All pools are sub pools of the main server pool `pglobal`. A connection pool is a sub pool of the child server pool handling the connection and all request pools are sub pools of the corresponding connection pools.

Internal structure

Internally, a pool is organized as a linked list of sub pools, blocks, processes and callback functions. Figure 4.25 gives a simple view of a block and an example of a linked list of blocks.

If memory is needed, it should be allocated using predefined functions. These functions do not just allocate the memory but also keep references to that memory to be able to deallocate it afterwards. Similarly processes can be registered with a pool to be destroyed upon death of the pool. Additionally each pool can hold information about functions to be called prior to destroying all memory. That way file handlers and as such sockets can be registered with a pool to be destroyed.

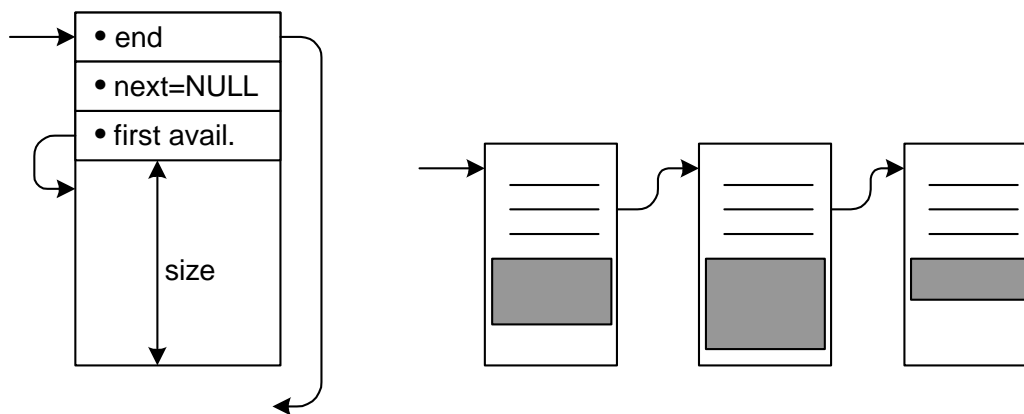


Figure 4.25: Linked List of Blocks

Termination sequence

When a pool is destroyed, Apache first calls all registered cleanup functions. All registered file handlers and that way also sockets are closed. After that, a pool starts to terminate all registered and running processes. When finished with that, all blocks of memory are freed. Usually Apache does not really free the memory for use by any other program but deletes it from the pool and adds it to a list of free memory that is kept by the core internally. That way the costly procedure of allocating and deallocating memory can be cut down to a minimum. The core is now able to assign already allocated memory to the instance in need. It only needs to allocate new memory if it has used up all memory that was deallocated before.

Blocks and Performance

Apache allocates memory one block at a time. A block is usually much bigger than the memory requested by modules. Since a block always belongs to one pool, it is associated with the pool once the first bit of memory of that block is used. Subsequent requests for memory are satisfied by memory left over in that pool. If more memory is requested than left over in the last block, a new block is added to the chain. The memory left over in the previous block is not touched until the pool is destroyed and the block is added to the list of free blocks.

Since blocks are not deallocated but added to a list of free blocks, Apache only needs to allocate new blocks once the free ones are used up. That way the performance overhead is heavily reduced, as the system seldom needs to be involved in memory allocation. Under most circumstances also the amount of memory used to store the same amount of information is smaller compared to conventional memory allocation, as Apache always assigns as much memory as needed without a lower limit. The size of a block can be configured using the Apache configuration.

4.6.3 The pool API

Since each pool handles the cleanup of resources registered with it itself, the necessary API functions are mainly used to allocate resources or to register already created resources with

a pool for cleanup. However, some resources can be cleaned up manually before the pool is destroyed.

Allocate memory

When allocating memory, developers can use two different functions. `ap_palloc` and `ap_pccalloc` both take the pool and the size of the memory needed as arguments and return a pointer to the memory now registered and available to use. However `ap_pccalloc` clears out the memory before returning the pointer.

The function `ap_strdup` is used to allocate memory from a pool and place a copy of a string in it that is passed to the function as an argument. `ap_strcat` is used to initialize a new allocated string with the concatenation of all string supplied as arguments.

Register callback functions

Basically any resource can be registered with a pool by supplying callback functions. Here the function which is to be called to free or terminate the resource and the parameters are registered. Upon the end of the lifetime of a pool these functions are called. That way any type of resource can make use of the pool technology.

For file descriptors and as such sockets, as these are file descriptors, the core offers equivalents to the `fopen` and `fclose` commands. These make use of the callback function registration. They register a function that can be called to close file descriptors when the pool is destroyed.

Process management

Additionally, a pool can manage processes. These are registered with the pool using the `ap_note_subprocess` function if the processes already exist. The better way is to use `ap_spawn_child` as that function also automatically registers all pipes and other resources needed for the process with the pool.

Sub pools

Sub pools are the solution if an existing pool is not suitable for a task that may need a large amount of memory for a short time. Sub pools are created using the `ap_make_sub_pool` function. It needs the parent pool handed over as argument and returns the new pool. Now this pool can be used to allocate resources relatively independent from the parent pool. The advantage is that the pool can be cleared (`ap_clear_pool`) or even destroyed (`ap_destroy_pool`) without affecting the parent pool. However when the parent pool the architecture shown in [fig: Internals: preforking BD] is destroyed, the sub pool is destroyed automatically.

Appendix A

Operating System Concepts

A.1 Unix Processes

Multitasking means concurrent execution of programs. A task can be a process or a thread, depending on the operating system. As there are usually more tasks than hardware processors in a computer system, the operating system has to multiplex the resources (processor, memory and I/O) to the tasks. There are various strategies for scheduling. In the following, we will focus on preemptive multitasking, which means that a task has no influence on how long it can keep the resources exclusively (and therefore can't block the system).

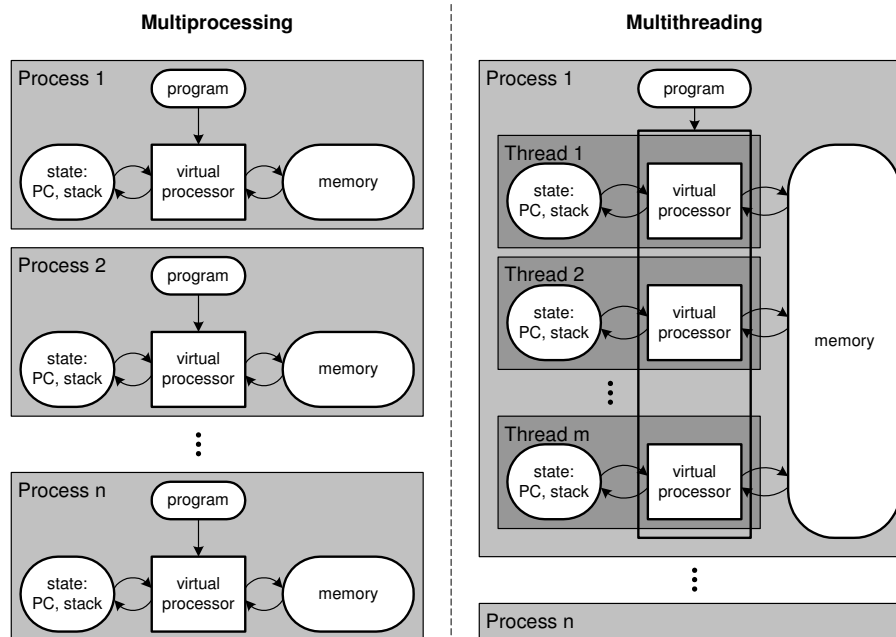


Figure A.1: Multiprocessing and Multithreading

Imagine a task like a virtual computer — it offers a CPU, memory and I/O. The state of a task can be found in the processor registers (for example the Program Counter PC) and the stack (the return addresses, the parameters of the procedure calls and local variables). The difference between processes and threads, as shown in figure A.1 is the isolation: While each process has an isolated memory space for its own, all threads (of a process) share one

memory space (including the program, of course). Threads are usually bound to processes which define the resource set shared by the threads. In the following, we will focus on UNIX processes.

A.1.1 `fork()`

In Unix processes are created using the system call `fork()`. The first process of a program is created upon the start of the program. Using `fork()` an additional process can be created. Fork simply creates a copy of the current process including all attributes. Like its parent, the new process continues with the code that follows the `fork()` instruction. To distinguish parent from child, the return value of the fork call can be used. Fork will return 0 to the newly created process while the parent gets the process id of the child process. Therefore the fork system call is usually followed by a decision based on fork's return value.

The example code below is a shortened extract of the `make_child` procedure of the Apache 2.0 preforking MPM:

```
if ((pid = fork()) == -1)
{
    /* This path is only entered
     * when fork did not succeed */

    ap_log_error(APLOG_MARK, APLOG_ERR, errno, s,
                 "fork: Unable to fork new process");

    /* fork didn't succeed. Fix the scoreboard
     * or else it will say SERVER_STARTING
     * forever and ever */

    (void) ap_update_child_status_from_indexes(slot,
                                                0, SERVER_DEAD, (request_rec *) NULL);

    /* In case system resources are maxxed out,
     * we don't want Apache running away with the
     * CPU trying to fork over and over and
     * over again. */

    sleep(10);
    return -1;
}

if (!pid) {
    /* In this case fork returned 0, which
     * means this is the new process */

    apr_signal(SIGHUP, just_die); // It registers a
    apr_signal(SIGTERM, just_die); // for a few
                                   // signals
}
```

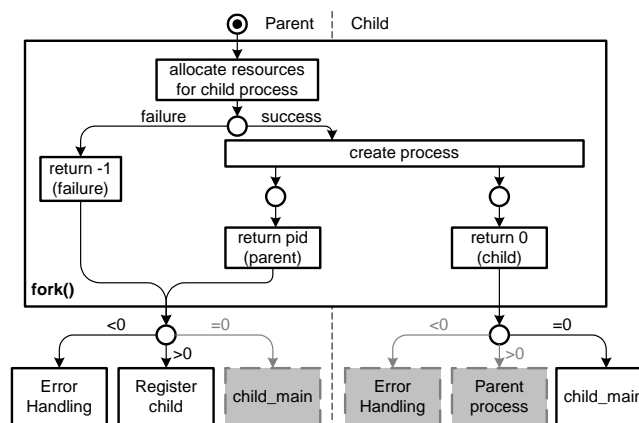
```

    child_main(slot); /* And then calls the main
                       * routine for child worker
                       * processes, which it will
                       * never return from. */
}

/* As the child never returns only the parent process
 * will get to this stage and only if the fork call
 * succeeded. The variable PID contains the process
 * ID of the newly spawned process, which it write to
 * the scoreboard, a table used to register processes */

ap_scoreboard_image->parent[slot].pid = pid;
return 0;

```

Figure A.2: Internals of the `fork()` system call

Look at figure A.2 for a petri net version of the source code above. Looking at the transitions at the bottom we see that the program provides three possible cases depending on the return value of `fork()`. Some of them are grey because they will never be executed at all, depending on whether the parent or the child process executes the code.

A.1.2 `exec()`

When a process wants to execute a different program it can use any of the `exec()` system calls. Exec discards all data of the current process except the file descriptor table — depending of the variant of `exec` this in- or excludes the environment — and loads and starts the execution of a new program. It can be combined with a prior `fork()` to start the program in a new process running concurrently to the main process.

All variants of `exec()` serve the same purpose with the exception of the way the command line arguments are submitted. The different `exec` calls all expect command line arguments using different data types which gives flexibility to the programmer. Additionally there are separate calls that can be used when a new environment is desired for the program that is to be executed. Otherwise the program will simply use the existing environment of the calling process.

The different exec calls are:

	argument list	argument vector
keeping environment	<code>execl</code>	<code>execv</code>
keeping env, using PATH	<code>execlp</code>	<code>execvp</code>
new environment	<code>execle</code>	<code>execve</code>

Execs demanding an argument list expect multiple arguments each containing a single argument to the program called. Execs demanding a vector demand an array of arguments just like the `**argv` array that each main method expects.

A.1.3 `wait()` & `waitpid()`

The system calls `wait()` and `waitpid()` are used to synchronize a parent process with the termination of its child processes. The parent blocks until a child process dies. The difference is that `wait()` waits for the termination of any child process while `waitpid()` allows to specify the child process by its PID. The wait system call has some closely related variants, `wait3()` and `wait4()` which allow to gather more information about the process and its termination and to supply options for the waiting.

A.1.4 `kill()`

The `kill()` system call is used to send signals to processes. Even though the name implies that the `kill()` system call is solely used to kill processes, more is behind it. However, a signal that is neither caught nor ignored terminates a process which is why this system call is called `kill`. The syntax is `kill(int pid, int signal)`. Most signals can be caught or ignored, however the signal `SIGKILL` (#9) cannot and will terminate the process in any case. For more information on signals and the `kill()` system call look at the next section.

A.2 Signals and Alarms

A.2.1 Signals

Signals are a way of inter-process communication (IPC) that works like an interrupt. To send a signal to a process, a program uses the system call `kill()`. A process receiving a signal suspends execution of its program immediately and executes a signal handler procedure instead. After that it resumes the program. A process can register a signal handler procedure for each individual signal type. If no signal handler has been registered for the issued signal, the default handling is to terminate the process.

A signal handler is a procedure which has to be registered for one or many signals and which is restricted in its capability to execute operating systems calls. Therefore a signal handler will either react to the signal directly or save necessary information about the signal, so that the main program can handle it at a later point. The Unix version of Apache uses a very simple signal handler (see figure 4.8 right-hand side) which just sets flags.

The process can also *ignore* every signal except `SIGKILL` which will always terminate the addressed process. Any signal which is ignored will simply be discarded. Another option is



to *block* a signal, which is similar to ignoring the signal. The signal will then not be relayed to the process as if it was ignoring it. However any arriving signal is saved and will be forwarded to the process when the signal is unblocked. Using that feature a process can prevent a signal from interrupting its execution during important parts of the execution sequence. Blocking signals at the beginning of a signal handler and reenabling at the end of its execution is a good way to prevent race conditions of multiple signal handlers executing concurrently.

Additionally most UNIX flavours allow associating single signal handlers with a *set of signals*. Signal sets can be ignored or blocked as a whole set. When a signal handler is responsible for a set of signals, the parameter of the handling function will supply information about which signal triggered the handler.

However, signals are a technique that is very limited. Only a fixed range of signals exist. Apart from information that might be encoded in the choice of a specific signal type there is no possibility to send additional information to the addressed process. Furthermore the routines that handle signals are limited in their capabilities. Only a narrow range of system calls may be executed.

A.2.2 Usage

signal() and kill()

Signal handlers are registered for a specific signal type using the `signal()` system call. `signal()` expects an integer for the signal type and a function pointer to the signal handling procedure. Generally a signal can be sent using the `kill()` system call. This function requires a process id and a signal type as arguments. On most UNIX systems there is also a command that can be used on the shell named `kill`. Users can use it to send signals to running processes. Unix versions of Apache can be restarted or shut down using the `kill` command.

alarm() and raise()

A process can address signals to itself using the `raise()` call. `Alarm()` instructs the system to interrupt the process using the `SIGALRM` signal after a certain time, like an alarm clock. Therefore the `alarm()` system call only be used by a process to send a specific signal to itself.

Signal Masks

Sets of signals can be managed easily by using so-called signal masks. A set of signals can be assigned to one single signal handler or a can be equally ignored or blocked by a process.

A.3 Sockets

The technology called sockets is used to enable high level applications and services to communicate using the network hardware and protocol services offered by the operating system. This section will provide information about sockets and TCP/IP covering both UDP

and TCP. Sockets are also used for other protocols like Netware's IPX/SPX. The examples in this section will be restricted to TCP/IP.

A.3.1 TCP vs. UDP Ports

TCP/IP offers two types of ports software systems can use to communicate: TCP ports and UDP Ports.

TCP (Transmission Control Protocol) offers a reliable end-to-end connection using an unreliable network. UDP (User Data Protocol) is a connectionless transport protocol used to send datagram packets. Although unreliable, it is fast because of the small protocol overhead.

A.3.2 Asynchronous Events

Network traffic by design is asynchronous. That means a software system and especially a service do not have influence on when traffic arrives. Although most operating systems today engage multitasking technologies, it would be a waste of resources to let each task poll for incoming events continuously. Instead, the operating system ensures to capture all traffic of a network interface using asynchronous event technologies such as hardware interrupts. Tasks use a blocking operating system call to wait for incoming data — blocking means that the tasks remains suspended until the event it waits for occurs.

A.3.3 Sockets in general

A socket can be seen as a mediator between the lower level operating system and the application. Roughly the socket is a queue. The higher-level application asks the operating system to listen for all messages arriving at a certain address, to push them into the socket queue and to notify the caller after data has arrived. To remain synchronous, operating system calls like `listen()`, `accept()`, `receive()`, and `recvfrom()` are blocking calls. That means the operating system puts the thread or process to sleep until it receives data. After receiving data and writing it into the socket queue, the operating system will wake the thread or process. The abstract system structure of a set of agents communicating via sockets is shown in figureA.3.

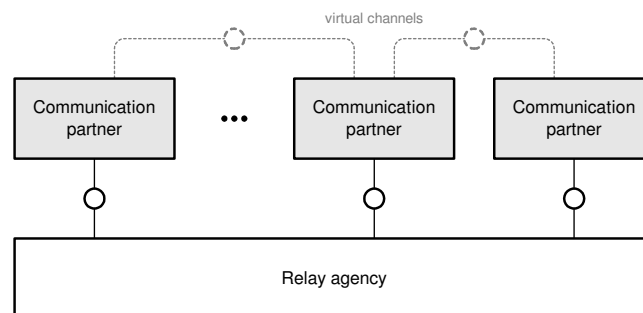


Figure A.3: Basic concept of sockets

Figure A.4 shows a detailed view. Sockets are managed by a socket communication service, which provides the socket API to the communication partners, i.e. the different application components. Using this API it is possible to setup and use sockets without any knowledge about the data structure of sockets and the underlying protocols used for communication.

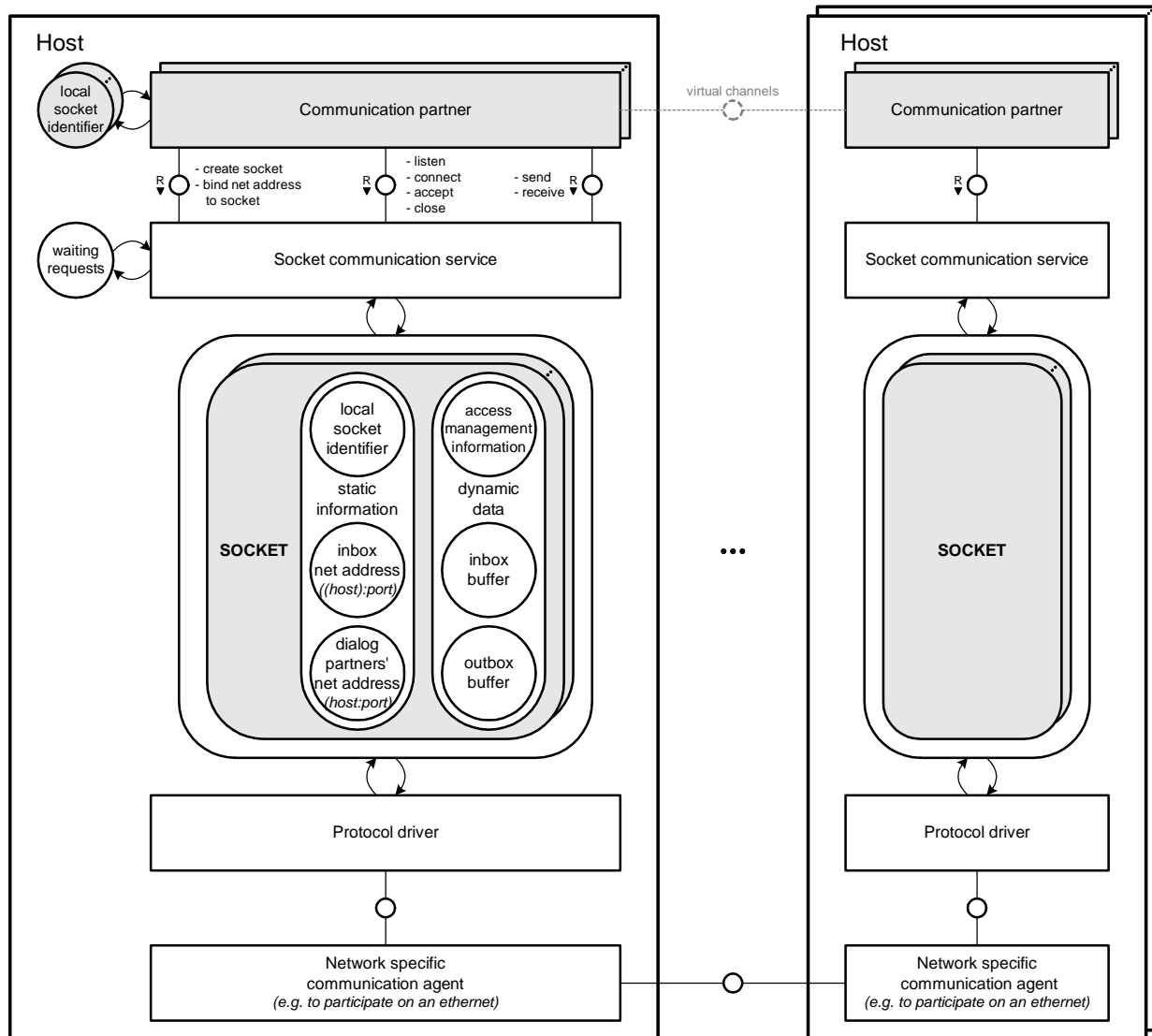


Figure A.4: Communication via sockets

A.3.4 TCP Sockets

The TCP protocol supports connection based communication. So before communication is possible a connection has to be established using a distinguished setup channel. That's why, with TCP, a service waiting for incoming connections has to employ two different types of sockets. One socket type has to be used for receiving connection requests and establishing connections. After a connection request arrived, another socket type has to be used for data transfer with the entity that issued the request. The client uses the same type of socket to establish a connection.

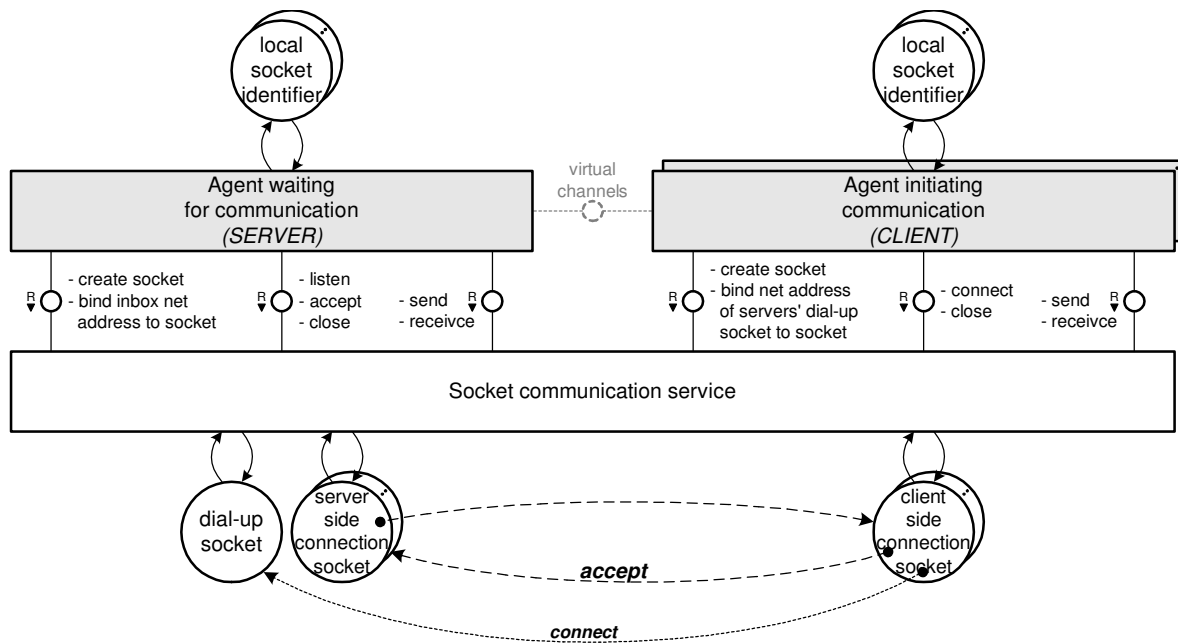


Figure A.5: Dialup and connection sockets

The first type, generally referred to as the “*dialup socket*” or “*server socket*” or “*listen socket*”, is bound to one or more local IP addresses of the machine and a specific port. After a connection was requested by a remote application, a new socket usually called “*connection socket*” is created using the information of the packet origin (IP address and port) and the information of the local machine (IP address and port). This socket is then used for communication via the TCP connection. Figure A.5 shows the different kinds of sockets and the operating system calls to use and control them. Figure A.6 illustrates the relation between ports and connection resp. dial-up sockets.

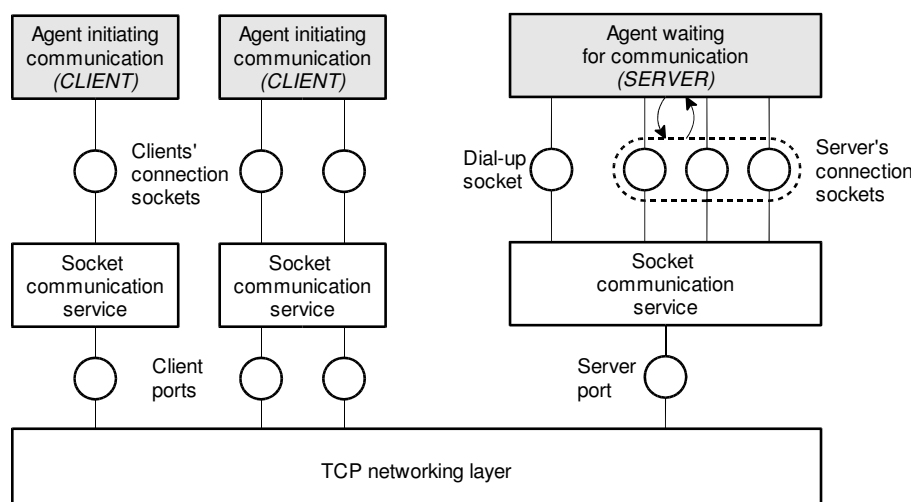


Figure A.6: Sockets and ports

Basically, a listen socket only contains information about the local port and IP address, it is therefore only used to listen, as the operating system would not know where to send infor-



mation. The connection socket contains both origin and local information and can therefore be used to transmit and receive data.

There may be multiple TCP sockets (one listen socket and an undefined number of connection sockets) for each port. The operating system decides based on the information of the communication partner to what socket the received information has to be queued.

A.3.5 UDP Sockets

UDP Ports do not engage connection oriented techniques or safety mechanisms. Therefore a UDP socket is created using the local IP address and the port number to be used. Only one socket can be created for one port as a UDP socket is never associated with a connection partner. For sending, a host has to supply information about the communication partner; for receiving, it is optional.

In a multitasking environment multiple threads or processes that each serve a separate communication partner would, unlike in TCP, not have an own socket but share the socket. When receiving and transmitting they will then supply the address of their communication partner. As HTTP is based on TCP when used in TCP/IP networks, UDP is not used in Apache.

A.4 Pipes

In Unix based operating systems the primary means for inter-process communication is the pipe concept. A pipe is a unidirectional data connection between two processes. A process uses file descriptors to read from or write to a pipe.

A pipe is designed as a queue offering buffered communication. Therefore the operating system will receive messages written to the pipe and make them available to entities reading from the pipe.

The pipe is referenced using file descriptors for both end points. Each endpoint can have more than one file descriptor associated with it.

A.4.1 Using pipes for inter process communication

Most software systems use the functionality and properties of process creation using `fork()`. Fork copies program, memory and the file descriptor table of the parent to the new child process. When a parent process wants to create a process that it wants to communicate with afterwards, it will create a pipe prior to the process creation. When the new process is spawned, the parent process and all its properties are copied. Therefore the file descriptors containing references to the pipe are copied as well. Now both processes have references to the input and the output end of the pipe. If one process only uses the input end and the other only uses the output end of the pipe, a one-way communication is established between both processes. As pipes can transmit any data as a stream, a pipe can be used to transmit data or simply to synchronize operations between both processes. If two-way communication is desired, two pipes have to be used.

A.4.2 STDIN, STDOUT, STDERR

By default, each process has three file descriptors when it is started. These file descriptors are called `STDIN`, `STDOUT` and `STDERR`. As the names let predict `STDIN` is the standard means to receive data, `STDOUT` is used to output data and `STDERR` is used to transmit error messages. By default these file descriptors are all connected to the shell via pipes that executed the program. A very popular way to establish communication with a child process is to simply redirect the `STDIN` and `STDOUT` to a pipe that was created by the parent process prior to spawning the child. However the main advantage is that the newly created process with its redirected pipes can execute a separate program in its own environment using `exec()`. Although `exec()` resets the process' memory and loads a new program it keeps the file descriptor table. This program then inherits the process' `STDIN` and `STDOUT` which enables communication with the parent process which is then able to interact with the executed external program.

In Apache this is used for external script interpreters. Figure A.7 shows the static structure of such a pipe setup after the `STDIN` and `STDOUT` of parent and child have been redirected to a pipe. Another pipe would be needed to establish two-way communication.

A.4.3 Implementation

Pipes are created using the `pipe()` system call. `Pipe()` returns an array of file descriptors of size 2. These array elements are used to point to the input and the output end of the pipe. After the successful execution of the pipe function, the pipe can be used for input or output or it can be redirected to any of the standard file descriptors. To achieve that the standard file descriptor in question has to be closed and the other pipe has to be redirected to the standard descriptor using the function `dup()`, which expects a single integer, the file descriptor identifying the pipe as an argument. `Dup` actually doubles the output or input to the standard pipe and the supplied argument. But as the standard pipe was closed before calling `dup`, only the new pipe will be used. By closing all file descriptors of the pipe using `close()`, the pipe can be closed as well.

A.4.4 Named pipes

A drawback of regular pipes lies in the fact that the communication partners have to share file descriptors to get access to the pipe. This can only be done by copying processes with `fork()`.

A named pipe (FIFO) can be accessed via its name and location in the file system. Any process can get access to a named pipe if the access rights of the named pipe allows it to. A named pipe is a FIFO-special file and can be created with the `mknod()` or `mkfifo()` system call.

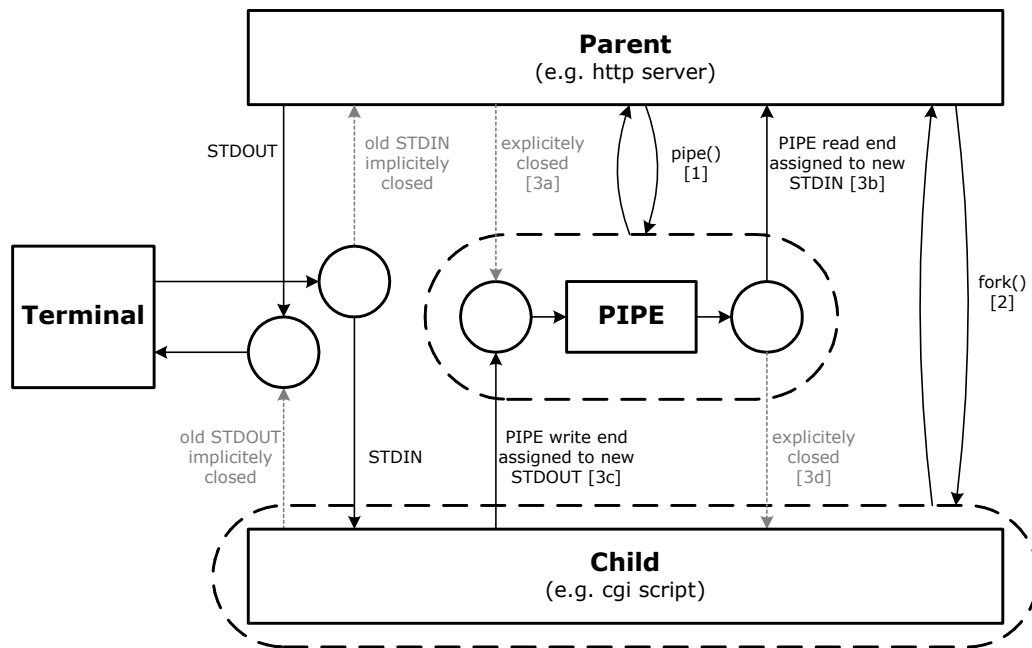


Figure A.7: Pipes

A.5 Longjmp

A.5.1 Concept

Longjump is a programming concept used in C to manipulate the flow of the execution sequence. Roughly it can be described as setting a mark by saving the current state of the processor of a program - in this case a thread or a process. After the state has been saved it can be restored at a further point in the sequence. To understand this concept it is essential to know that each process or thread has a stack for subroutine calls. Whenever the processor executes a subroutine call, it saves the program counter to the stack together with the parameters for the procedure, local variables are also put on the stack. At the end of a subroutine, the processor reads the saved program counter and discards all data put on the stack after this entry. The program continues with the next statement. The Longjump concept uses the fact that the stack contains this information.

When calling the function `setjmp()` the current processor's registers including the stack pointer is saved with the mark. Therefore the mark can be used to determine at which position the execution continues after `setjmp()`. When at any following point in the program execution the `longjmp()` function is used with the mark as an argument, the processor's registers including stack pointer is set to the position where it was before. The program will then continue as if the `setjmp()` call returned again. Only the return value of this call will be different. Therefore the Longjump concept is very similar to using "goto". However as opposed to `longjmp()`, "goto" will not restore any of the program's state. Therefore `longjmp()` can be used for jumps without corrupting the call stack. The drawback of using Longjump is that one has to be aware of the underlying systems technology to fully understand the concept, and of course that the program becomes more fuzzy as non standard programming concepts are used. A typical application for Longjumps is exception handling.

A.5.2 Example Longjump

In this example a single thread or process is considered. At a certain point of the execution sequence the function `sub1()` is called. As described above a new element containing information about the calling routine is put on the stack. During the execution of `sub1()`, `setjmp()` is called to mark the current position of the execution sequence and to save the state information. `setjmp()` returns 0 as it is called to mark the current position. Therefore the execution continues in the normal branch of the execution sequence. `sub1()` then calls `sub2()` which will call `sub3()`. Within `sub3()` the `longjmp()` system call is issued. `Longjmp()` requires two parameters. The first is the mark, used to define the position to jump to. The second parameter is the return value that the imaginary `setjmp()` call will return when being jumped to using the `longjmp()` call.

The execution will continue with the instruction following the `setjmp` call. That will usually be a decision based on the return value of `setjmp` to be able to differentiate whether `setjmp()` was just initialised or jumped to using `longjmp()`.

The code for the example above that illustrates the use of Longjump and also resemble the petri net of figure A.8 is:

```
void sub3 (jmp_buf buffer)
{
    longjmp (buffer,1); /* Jump to the point in sequence
                        * which buffer marks and have
                        * setjmp return 1 */
}

void sub2(jmp_buf buffer)
{
    sub3(buffer); /* call another sub function to fill up the
                  * stack*/
}

int sub1()
{
    jmp_buf buffer; // we can save a stack point in here
    if (setjmp(buffer))
    {
        // when setjmp is jumped to we will end up here
        exit(0);
    }
    else
    {
        /* when setjmp is called to save the stack
         * pointer we will enter this branch */
        sub2(buffer); // call any sub function
    }
}
```

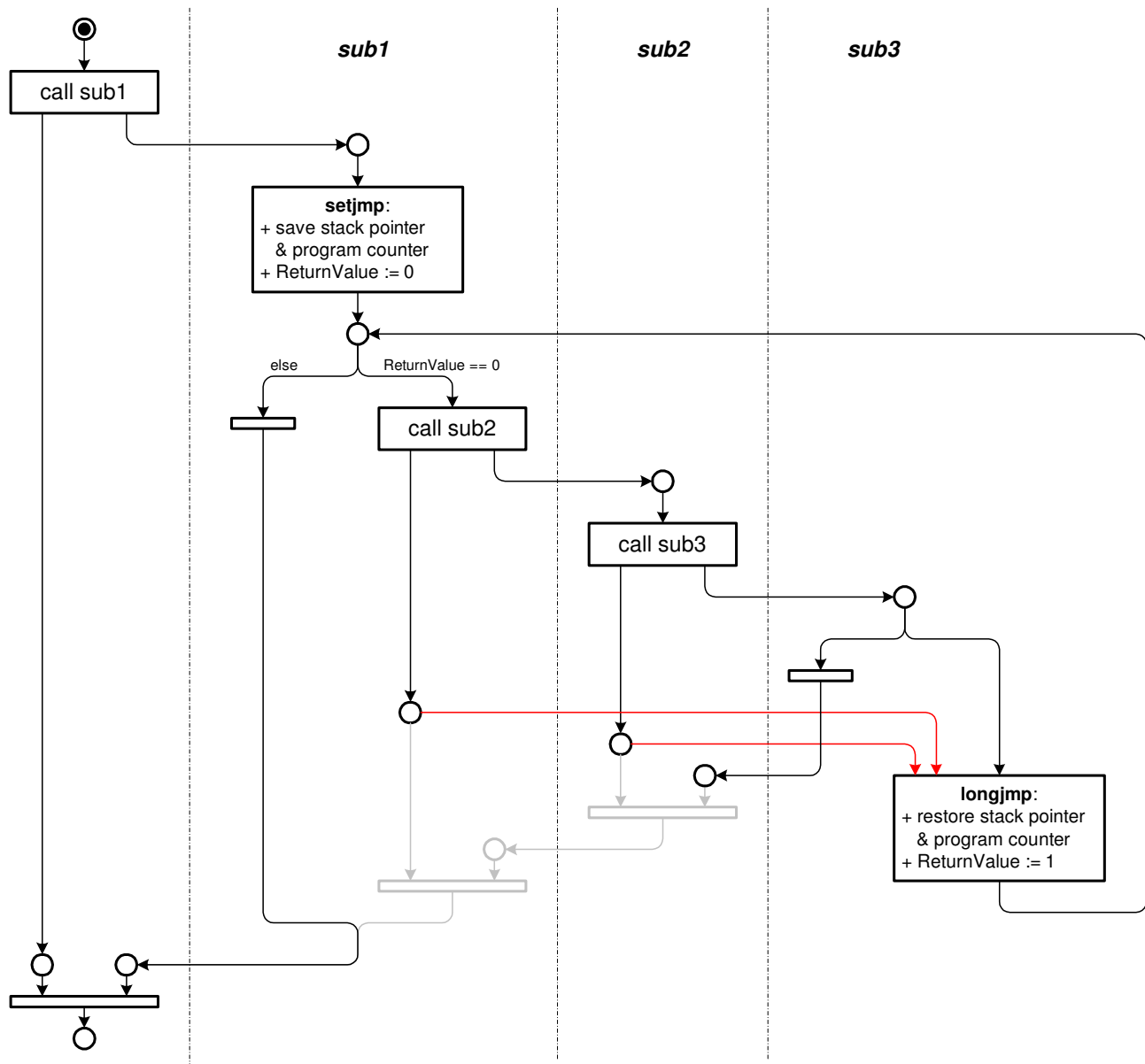


Figure A.8: longjmp behaviour

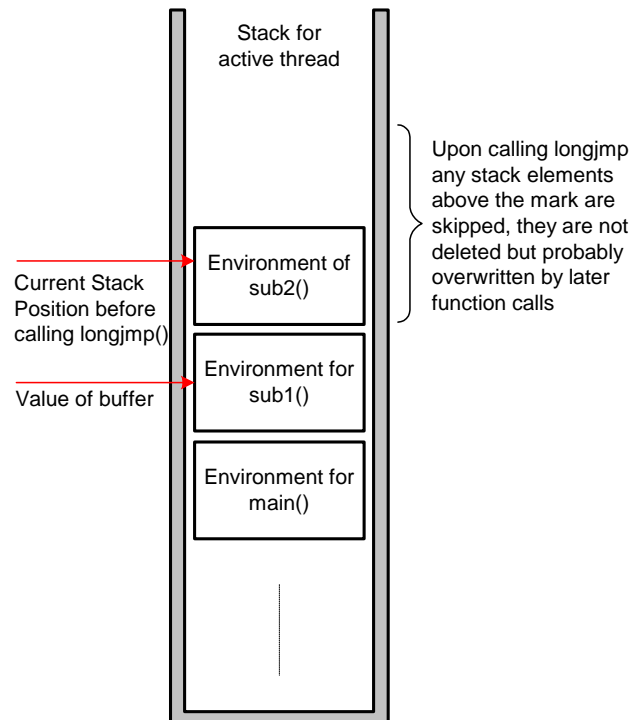


Figure A.9: Stack content before calling longjmp() in the example above

A.5.3 Common application

One of the major uses for the Longjump concept is exception handling. Usually when using Longjump for exception handling the program will save a mark using `setjmp()` right at the beginning of the execution sequence. It is followed by a decision that will branch to the normal program execution sequence based on the return value of `setjmp()`. A return value of 0 indicates that `setjmp()` was initially called to save a mark and execution is therefore continued normally. However, if the return value of `setjmp()` indicates that `setjmp()` was jumped to then it usually includes information about the type of the exception that occurred. The corresponding exception handling routine can therefore be started. The advantage of such error handling is its centralization and that it can be used from any point in the program execution sequence as the mark was saved at the lowest point of the subroutine execution hierarchy. Implementing exception handling this way very much resembles exception handling in higher level languages like Java and C++. The difference is that in C++ and Java exceptions can be caught in any procedure along the hierarchy of procedure calls. To implement such behaviour via Longjump multiple marks and additional logic to decide to which mark to jump to is required.

While exception handling is the major application area for this concept, Longjump can also be used to maximize performance in skipping multiple subroutine returns. A recursive algorithm for example might need a large amount function calls to compute the final result within the deepest function of the hierarchy. There the result is known but multiple function returns are needed to transfer the result back to the function originally triggering the recursion. In this case one single longjump can transfer the result by simply skipping the returns and restoring the original stack pointer that was current before the recursive algorithm was started.



Longjump can be used to jump within different stacks or even stacks of different threads. In an environment where multiple stacks exist to keep track of execution sequences - a stack might have been manually created or multiple threads that all have an own stack exist - Longjump can be used to set the stack pointer of a thread to a stack that originally does not belong to that thread. This can be used to implement a scheduling algorithm within a process. However modeling that kind of behaviour is rather difficult and does have many pitfalls. As you can imagine using one thread to jump to a stack position of a thread that is still active can corrupt the second thread's stack and will likely lead to undetermined behaviour or crashes.

A.5.4 Common Pitfalls

Even though the Longjump concept is a very powerful tool, it still demands a very detailed understanding of the operating system concept it is based on. Again different operating systems differ and might not support the functionality equally. Different flavours of Unix or Linux handle concurrency differently and some save more information on the stack than others. The existence of `siglongjmp()` function that has the same behaviour as `longjmp` with the addition of making sure that also the current signalling settings are saved and restored underlines that fact.

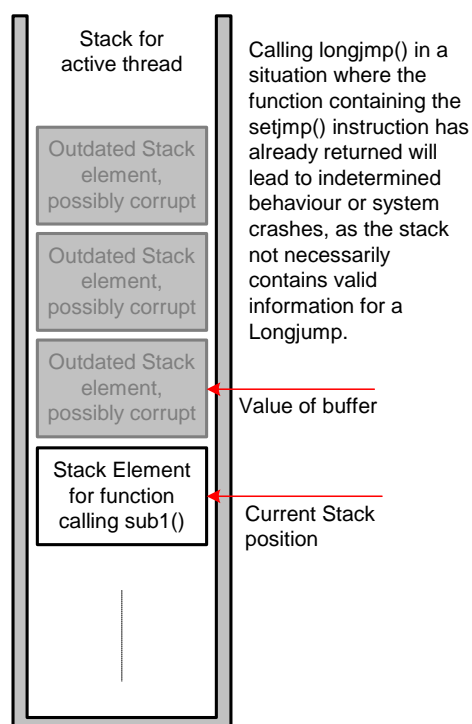


Figure A.10: Calling `longjmp()` can cause errors when used without caution

Additionally certain thread properties, like open file descriptors are simply not stored in the stack and can therefore be lost when jumping to a different thread. Local variables on the other hand are stored on the stack. When jumping from a point in the program sequence that involves any local pointer variables the developer has to make sure that no memory is still allocated before `longjmp()` is called to avoid memory leaks, as these pointer variables

will not exist at the destination of the jump, but the memory will remain allocated until the program terminates.

The stack pointer value stored at any point of the programs execution should not be used any further after the current stack pointer has pointed or still points to a lower stack element than the marked one. At such a point the procedure that contained the setjmp call has already returned. The stack pointer was or is set to a lower level and any intermediate subsequent procedure call might have overwritten the element the saved pointer value originally pointed to. Therefore undetermined behaviour or crashes are likely to occur.

In the end Longjump is a very powerful tool when used with great care. On the other hand the excessive use of longjump can create very confusing execution sequences and procedure call hierarchies that are hard to model and keep track of. The code will also become harder to read the more Longjump is used.

Appendix B

Sources

B.1 Simple HTTP Server

```
1  #!/usr/bin/env python
2
3  # simple HTTP Server in Python 2
4
5  import socket, string, os
6  from stat import *
7
8  # Please change setting to your needs
9  PORT = 8081 # Arbitrary non-privileged server
10 DOCUMENT_ROOT = "Test-HTML/"
11
12 # procedure process_request
13 def process_request( clientsocket, address ):
14
15     # read HTTP request
16     # Define Response Headers
17     responsePrefix = "HTTP/1.0 200 OK\nServer:HPIServer/1.0\n"
18     response_NotSupported = "" "HTTP/1.0 400 Bad Request
19 Server: HPIServer/1.0\nConnection: close
20 Content-Type: text/html; charset=iso-8859-1\n
21 <HTML><HEAD><TITLE>400 Bad Request</TITLE></HEAD>\n"" "
22     response_FileNotFound = "" "HTTP/1.0 404 Not Found\nServer: HPIServer/1.0
23 Connection: close\nContent-Type: text/html; charset=iso-8859-1\n
24 <html><head><title>File not found</title></head>\n"" "
25
26     # Create File descriptor for request to read the first line of the request
27     clientsockfile = clientsocket.makefile('rw')
28     requestline = clientsockfile.readline();
29
30 # process GET method only
31     requestwords = string.split( requestline )
32     if requestwords[0] == 'GET':
33
34 # search file
35     try:
36         if requestwords[1] == '/':
37             # assume index.html was requested (no directory view)
38             requestwords[1] = '/index.html'
```

```

39     filestatus = os.stat( DOCUMENT_ROOT + requestwords[1] )
40     if S_ISREG(filestatus[ST_MODE]): # Regular file ?
41         # detect file size using stat()
42         filesize = filestatus[ST_SIZE]
43 # open file in binary mode
44     file = open(DOCUMENT_ROOT + requestwords[1], 'rb')
45     else:
46         raise OSError
47 except OSError:
48     clientsocket.send( response_FileNotFound )
49     clientsocket.send( "<body> File: " + requestwords[1] +
50                       " not found</body>\n</html>\n" )
51 else:
52
53 # Detect file's MIME type by suffix.
54     suffix = string.split( requestwords[1], '.' )[-1]
55     # text/html image/gif image/jpeg audio/midi application/java
56     if suffix == 'html': filetype = 'text/html'
57     elif suffix == 'gif' : filetype = 'image/gif'
58     elif suffix == 'jpg' : filetype = 'image/jpeg'
59     elif suffix == 'mid' : filetype = 'audio/midi'
60     elif suffix == 'class' : filetype = 'application/java'
61     else: filetype = "unknown"
62
63 # send response
64     # Send response header
65     clientsocket.send( responsePrefix )
66     clientsocket.send( "content-type: " + filetype + "\n" )
67     clientsocket.send( "content-length: " + repr(filesize) + "\n" )
68     clientsocket.send( "\n" )
69
70     # send file in 1024 byte chunks
71     while 1:
72         data = file.read(1024)
73         if not data:
74             break
75         clientsocket.send( data )
76     else:
77         clientsocket.send( response_NotSupported )
78 clientsocket.send( "<body> request: " + requestline +
79                  "</body>\n</html>\n" )
80
81     # close Filedescriptor for request
82     clientsockfile.close()
83
84
85 #-----
86 # Server main
87
88
89 # Initialization
90
91 # Open server socket
92
93     # inet Socket (not UNIX), Type Stream (not datagram)
94     serversocket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
95     serversocket.bind((socket.gethostname(), PORT)) # bind to Port PORT
96     # tell OS to activate port for a maximum

```



```
97     # of 1 connection simultaneouslyserversocket.listen(1)
98
99     # request-response loop
100    try:
101        while 1:
102            # wait for request and establish connection
103            ( clientsocket, address ) = serversocket.accept()
104            process_request( clientsocket, address )
105            # Close connection
106            clientsocket.close()
107    finally:
108        # Deactivation
109        serversocket.close()
110
111    • .
```

Appendix C

Fundamental Modeling Concepts (FMC)



This chapter gives an introduction to the Fundamental Modelling Concepts (FMC) developed by Siegfried Wendt and taught at the Hasso-Plattner-Institute (HPI) in Potsdam, Germany. For more information, look at the FMC web site <http://fmc.hpi.uni-potsdam.de>.

C.1 What is FMC?

Intention and background FMC is the acronym for the *Fundamental Modelling Concepts*, primarily a consistent and coherent way to think and talk about dynamic systems. It enables people to communicate the concepts and structures of complex informational systems in an efficient way among the different types of stakeholders. A universal notation, easy to learn and easy to apply, which originates from existing standards, is defined to visualize the structures to communicate in a coherent way. In contrast to most of the visualization and modeling standards of today, it focuses on human comprehension of complex systems on all levels of abstraction by clearly separating conceptual structures from implementation structures. FMC is based on strong theoretical foundations, has successfully been applied to real-life systems in practice (at SAP, Siemens, Alcatel etc.) and also is being taught in software engineering courses at the Hasso-Plattner-Institute of Software Systems Engineering.

Purpose of this document This quick introduction will give you an idea of what FMC is all about by presenting you the key concepts starting with a small but smart example. Following the example, you will learn about FMC's theoretical background, the notation and hopefully get a feeling about the way FMC helps to communicate about complex systems. At the first glance, you might find the example even trivial, but keep in mind that this little example presented here may be the top-level view of a system being realized by a network of hundreds of humans and computers running a software built from millions of lines of codes. It would hardly be possible to efficiently develop such a system without efficient ways to communicate about it.

Levels of abstraction The example describes different aspects of a travel agency system. Starting with a top level description of the system, we will shift our focus toward implementation, while still remaining independent from any concrete software structures. So

don't expect to see UML class diagrams, which doubtless might be helpful to represent the low-level structures of software systems.

C.2 Compositional structures and block diagrams

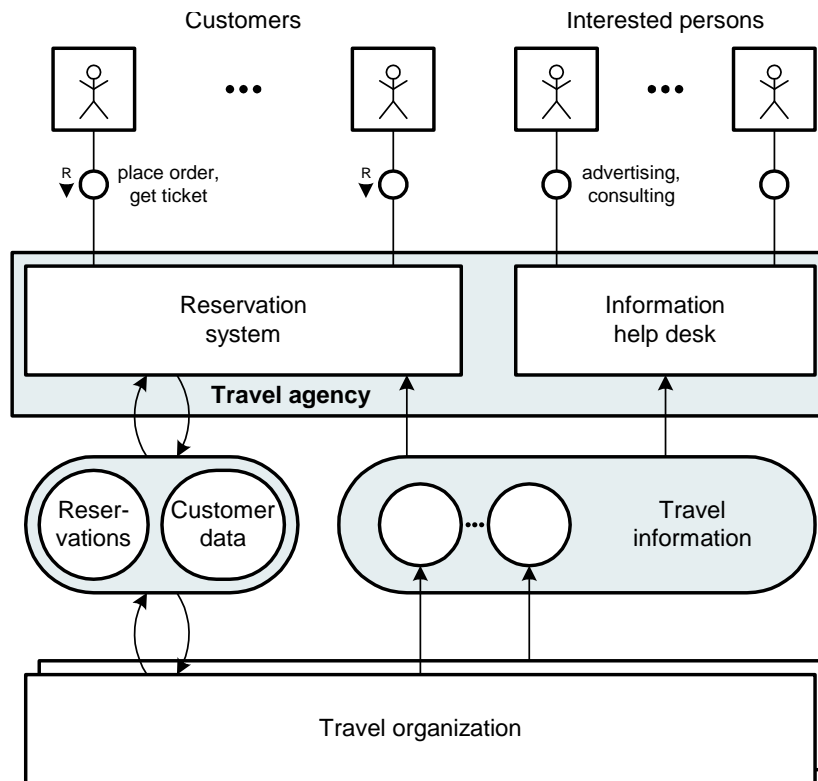


Figure C.1: Block diagram: Travel agency system

C.2.1 Example system

Figure C.1 shows a *block diagram* which represents a model of the *static compositional structure* of a travel agency system and its *environment*. In the upper part of the block diagram several rectangles are shown, each containing the stylized figure of a human. The left group are customers of the travel agency interacting with the reservation system. Reservation orders can be placed, which are transmitted to the corresponding travel organizations, while the customers are issued their tickets. To the right are those persons looking for travel information using the information help desk. This information is provided by the different travel organizations and is stored in a location symbolized by the big rounded rectangle labeled *travel information*.

C.2.2 Block diagrams

Building blocks The block diagram in figure C.1, like any block diagram, represents a real or at least imaginable real informational system. The system being described is not more

abstract than anything else we consider to be real. Looking at the system we see components, artifacts of our mind, which relate to tangible distinguishable physical phenomena, like an apple, a computer, our family or a TV station — entities which exist in time and space. So an informational system can be seen as a *composition of interacting components* called *agents*. Each agent serves a *well-defined purpose* and communicates via *channels* and shared *storages* with other agents. If an agent needs to keep information over time, it has access to at least one storage to store and retrieve information. If the purpose is not to store, but to transmit information, the agents are connected via channels.

Agents are drawn as rectangular nodes, whereas locations are symbolized as rounded nodes. In particular, channels are depicted as small circles and storages are illustrated as larger circles or rounded nodes. Directed arcs symbolize whether an agent can read or write information from or to a storage.

Storage access In the example, the arcs directed from the nodes labeled "*travel organization*" to the storage node labeled "*travel information*" symbolize that the travel organizations *write* the travel information. Correspondingly the arc directed from the storage node labeled "*travel information*" to the agent node labeled "*information help desk*" symbolizes that the help desk *reads* the travel information. If an agent can modify the contents of a storage regarding its previous contents, it is connected via a pair of opposed bound arcs, called *modifying arcs*. The access of the reservation system and the travel organization to the customer data storage is an example.

Communicating agents If communication is possible in both directions, the arcs connecting the agents via the channel may be undirected. Looking the example, the communication between the help desk and the persons interested in some information is visualized that way. A very special, but common variant of this case is the request/response channels, where a client requests a service from another agent and after a while gets its response. To express, which side is requesting the service, a small arrow labeled with "R" for request and pointing from client to server, is placed beside the node symbolizing the channel. Examples are the channels between the customers and the reservation system.

C.3 Dynamic structures and Petri nets

C.3.1 Behavior of dynamic systems

Informational systems are dynamic systems. By looking at the channels and locations used to store, change and transmit information for some time, their *behavior* can be observed. Petri nets are used to visualize the behavior of a system on a certain level of abstraction.

Figure C.2 shows a Petri net describing the causal structure of what can be observed on the channel between the travel agency and one of its customers in our example. Buying a ticket starts with the customer ordering a ticket. Then the travel agency checks the availability and in case this step is successful, concurrently a ticket may be issued to the customer and payment is requested. The customer is expected to issue the payment and when both sides have acknowledged the receipt of the money respectively the ticket, the transaction is finished.

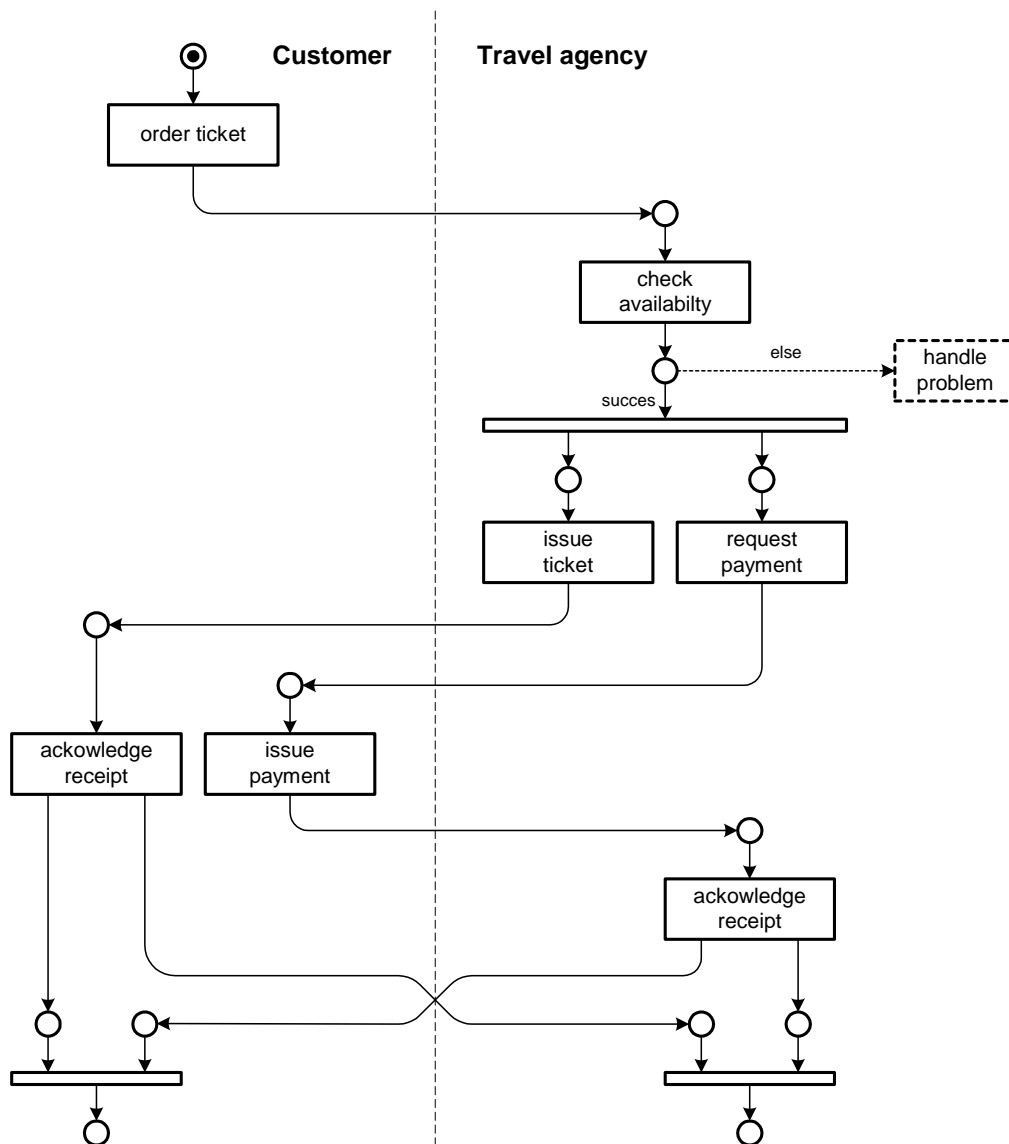


Figure C.2: Petri net: Buying a ticket

C.3.2 Petri nets

Building blocks Petri nets describe the causal relationship between the operations, which are performed by the different agents in the system. Each rectangle is called a *transition* and represents a certain *type of operation*. The transitions are connected via directed arcs with circular nodes called *places*. Places can be empty or marked, which is symbolized by a black token. The behavior of the system can now be simulated by applying the following rule to the Petri net: Whenever there is a transition with all its input places being marked and all its output places being unmarked, this transition may *fire*, meaning the operation associated with the transition is performed. Afterward all input places of the transition are empty and all its output places are marked.

So, looking at the Petri net shown in figure C.2, in the beginning only the transition labeled "order ticket" may fire. This means the first operation in the scenario being described will be the customer ordering a ticket. Because only the initial marking of a Petri net may be shown in a printed document, it is necessary to process the net by virtually applying the firing rules step by step, until you get an understanding of the behavior of the system. This is very easy, as long as there is only one token running through the net.

Patterns Common patterns are sequences of actions, loops and conflicts. A conflict is given if multiple transitions are ready to fire, which are connected with at least one common input place. Because the marking of that input place can not be divided, only one of the transitions may fire. In many cases, a *rule* is given to solve the conflict. In those cases *predicates* labeling the different arcs will help to decide which transition will fire. For example, different actions have to be taken depending on the outcome of the availability check. If the check was successful, the travel agency will issue the ticket and request payment.

Concurrency and Synchronization In our example, issuing the ticket and payment should be allowed to happen concurrently. Using Petri nets, it is possible to express *concurrency* by entering a state where multiple transitions may fire concurrently. In the example, we introduce concurrency by firing the unlabeled transition, which has two output places. Afterward both transitions, the one labeled "issue ticket" and the one labeled "request payment", are allowed to fire in any order or even concurrently. The reverse step is *synchronization*, where one transition has multiple input places, which all need to be marked before it is ready to fire.

Refinement Using refinement, it is possible to describe single transitions or parts of the Petri net in more detail. Also, by refining the compositional structure of the system or by introducing a new aspect, additional Petri nets may become necessary to describe the interaction of the new components.

C.4 Value range structures and entity relationship diagrams

C.4.1 Structured values and their relationships

Looking at dynamic systems, we can observe values at different locations which change over the time. In our model, agents, which have read and write access to these locations are responsible for those changes, forming a commonly static structure which is shown in a block diagram. Petri nets give us a visual description of the agent's dynamic behavior. To describe the structure and repertoire of information being passed along channels and placed in storages, we use *entity relationship diagrams*.

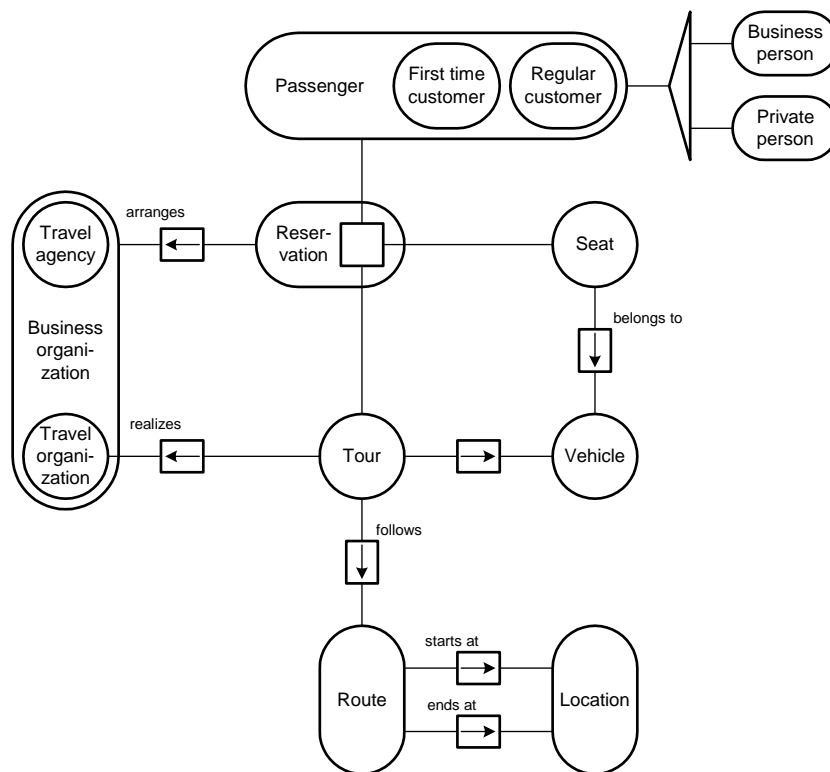


Figure C.3: Entity relationship diagram: Tour reservations

C.4.2 Entity Relationship Diagrams

Example Figure C.3 shows an entity relationship diagram representing the structure of the information, which is found when looking at the storage labeled "*reservations*" and "*customer data*" which both the "*reservation system*" and the "*travel organizations*" can access (see figure C.1). In the middle of the diagram, we see a rounded node labeled "*reservations*" which represents the set of all reservations being stored in system. Such a reservation is defined by a customer booking a certain tour, allocating a certain seat in a certain vehicle. The tour will follow a certain route, starting at some location and ending at some location. Looking at the passengers, first time customers are distinguished from regular customers. Independent from that, passengers can also be partitioned into business persons and private persons. The

system also stores information about the organization which has arranged a reservation and which travel organization realizes which tour.

Building blocks In entity relationship diagrams, round nodes visualize different *sets of entities* each being of a certain type. The sets in the example are passengers, business men, tours, vehicles etc. Each of them is defined by a set of *attributes* according to its type. Most elements of a set have one or more *relations* to elements of another or also the same set of elements. For instance, each route has one location to start at and one location to end at. Each relationship, i.e. each set of relations between two sets of entities being of a certain type, is represented by a rectangular node connected to the nodes representing the sets of entities participating in the relationship. So there is one rectangle representing the "start at" relationship and another representing the "ends at" relationship. Annotations beside the rectangle can be used to specify the predicate by an expression using natural language, which defines the relationship.

Cardinalities The *cardinality* of a relationship expresses the number of relations of a certain type one entity may participate in. Arrows respectively small numbers attributing the relationship nodes represent the cardinality. A bidirectional arrow symbolizes a one-to-one assignment of entities. A unidirectional arrow symbolizes that multiple elements of one entity set may be related to one single entity. Looking at our example, the arrow pointing from "seats" to "vehicles" expresses that every seat belongs to exactly one vehicle and that every vehicle contains multiple seats.

Partitions If one entity node contains multiple sub-nodes, this entity node represents the union of the entity sets being enclosed. Typically the elements of the union share a common type, an abstraction characterizing the elements of all subsets. For instance in the example "first time customers" and "regular customers" define the set of "passengers". But we can also distinguish "business persons" from "private persons", also the result of another true partitioning of "passengers". To avoid visual confusion caused by multiple containment nodes crossing each other, those unrelated partitions are symbolized using a longish triangle.

Objectification Sometimes it is helpful to interpret the elements of some relationship as entities itself, which by itself may participate further relations. Those abstract entities may have no direct physical counterpart. They are the *objectification* of some concrete fact, a statement about the relation among some given entities. A typical example is the relationship labeled "reservation" between the sets of "passengers", "tours" and "seats". Each element of that relationship embodies an entity itself — a reservation, which is arranged by some travel agency or travel organization.

Further application Entity relationship diagrams may not only be used to visualize the structure of the information stored in technical systems. They also can help to get some understanding of new application domains by providing an overview of the relations between its concepts.



C.5 Levels of abstraction

C.5.1 High-level structures

Show purpose of the system So far, the system has been described on a very abstract level only reflecting its purpose. The implementation of most components is still undefined. We see a high-level structure, which also could be explained with a few words. Looking at the block diagram (figure C.1), we only learn that the customers and interested persons are expected to be humans. Nothing is said about how the reservation system, the help desk, the travel organization are implemented, how the stored information looks like, whether it will be an office with friendly employees answering questions and distributing printed booklets or an IT system, which can be accessed using the Internet. All this is undefined on this level of abstraction.

System overview needed for communication Nevertheless, the model shows a very concrete structure of the example system. The system structure has been made visual, which highly improves the efficiency of communication. There is some meaningful structure you can point to, while talking about it and discussing alternatives. This would be inherently impossible if the real system did not exist yet or if the system just looked like a set of technical low-level devices, that could serve any purpose.

C.5.2 Hierarchy of models

By refining the high-level structure of the system while considering additional requirements, a hierarchy of models showing the system on lower levels of abstracting is being created. Again, the system description can be partitioned according to the three fundamental aspects, that define each dynamic system — compositional structure, behavior and value structures. Making the relationship between the different models visible using visual containment and descriptive text, comprehension of the systems is maintained over all levels of abstraction. Using this approach, it is possible to prevent the fatal multiplication of fuzziness, while communicating about complex structures without anything to hold on.

C.5.3 Lower-level structures

A possible implementation of the example system Figure C.4 shows a possible implementation of the information help desk and the storage holding the travel information. The storage turns out to be implemented as a collection of database servers, mail and web servers used by the different travel organizations to publish their documents containing the travel information. The information help desk contains a set of adapters used to acquire the information from the different data sources. The core component of the help desk is the document builder. It provides the documents assembled from the collected information and from a set of predefined templates to a web server. Persons being interested in the services from the travel agency may read the documents from this web server using a web browser. It is not obvious that the reservation system now has to request travel information from the information help desk instead of getting it itself. This is an example for non-strict refinement.

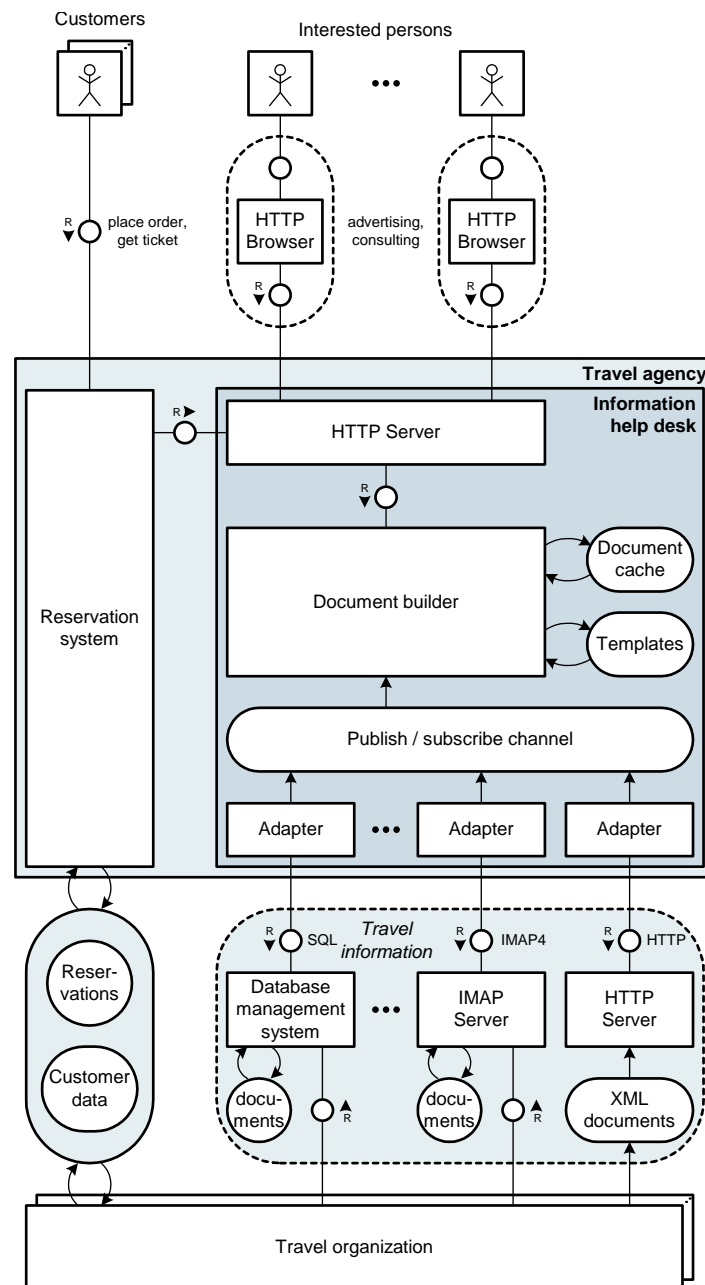


Figure C.4: Block diagram: Travel agency system



We could continue to describe the dynamics and value structures on that level, afterward refining or introducing new aspects one more time and so on. We won't do this here. When to stop this iteration cycle depends on the purpose of your activities: Maybe you are going to create a high-level understanding of the systems' purpose, maybe you are discussing design alternatives of the systems architecture or estimating costs, whatever.

Refinement of high-level structures Knowing the system's structure presented in figure C.1, it is quite easy to understand the more complex lower-level structure presented now. To ease comprehension, components from a higher-level system view should be projected into the lower-level system view whenever possible. In case implementation is done by simple refinement the result will be a containment hierarchy between the nodes representing components of different levels of abstraction. Without doubt, it would have been much harder to understand the example system if it would have been introduced on the level of adapters, mail servers and browsers. It would have been nearly impossible to create a common understanding without any figures at all.

C.6 What is special about FMC?

FMC focuses on *human comprehension of informational systems*. The key is the strict separation a very few fundamental concepts which can always be distinguished when communicating about informational system. Generally one should distinguish:

- *Didactic system models* serving the communication among humans versus *analytical models* serving the methodologically derivation of consequences
- *System structure* versus *structure of system description* (e.g. program structure)
- Purpose versus implementation

Essential about FMC is to distinguish:

- Compositional, dynamic and value range structure
- Active and passive components
- Control state and operational state

FMC provides the concepts to create and visualize didactic models. This enables people to share a common understanding of system structure and its purpose. Therefore FMC helps to reduce costs and risks in the handling of complex systems.

Appendix D

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000, 2001, 2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels)



generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, L^AT_EX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title Page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.

- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties — for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.



7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Bibliography

- [1] Ralf S. Engelschall. *Apache Desktop Reference*. Addison–Wesley, 2000.
<http://www.apacheref.com>.
- [2] The Apache Software Foundation. Apache http server project. Web site.
<http://httpd.apache.org>.
- [3] B. Gröne, A. Knöpfel, and R. Kugel. The apache modelling project. Web site.
<http://apache.hpi.uni-potsdam.de/>.
- [4] Douc MacEachern Lincoln Stein. *Writing Apache Modules with Perl and C*. O'Reilly, 1999.
- [5] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley and Sons, Ltd, 2000.
- [6] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

Glossary

- API** Application Programming Interface, a means to extend a software system in terms of functionality. Usually an API consists of a set of functions to extract data from the system and manipulate the system. In Apache, the core calls module callbacks via the module API while the procedures of a modul can use the Apache (core) API to access and modify Apache data structures and alter the way it handles requests. 100
- CGI** Common Gateway Interface, one of the first techniques of enhancing web servers and creating dynamic and multifunctional web pages. Using cgi the web server is enabled to execute applications or basic programming instructions which usually generate an HTML document. If used with HTML forms, the client transmits data with parameters encoded in the URL or with the message body of a "POST" request. Today other techniques like PHP, servlets, java server pages (JSP) or ASP are widely used alternatives for cgi. 25
- Directive (Configuration Directive)** Apache processes Configuration Directives while reading the configuration files during start-up or during request processing. 30
- Filter** A Filter processes data by reading from an input and writing to an output channel. A sequence of filters where one filter processes the output of another is called filter chain. Apache 2 uses an input filter chain to process the HTTP request and an output filter chain to process the HTTP response. Modules can dynamically register filters for both chains. 40
- Graceful Restart** Whenever the administrator wants to apply a changed configuration to a running Apache, he must restart the server. This would interrupt the server processes which currently process requests. A client would encounter an error or time-out. A graceful restart leaves alone all server processes which currently process a request. They can finish the answer to the request and terminate. Therefore a client does not notice the server restart. 67
- Handler** A handler is a callback procedure registered for a certain event. When the event occurs, the event dispatcher will call all handlers in a specific order. In Apache, the events during request processing are marked by hooks. Apache Modules register Handlers for certain hooks, for example for "URI translation". 35
- Header** Most messages transmitted in the internet consist of a header and a body. A header usually contains protocol information and meta information about the payload transmitted in the body. 11
- Hook** A hook is a processing step where the handlers (callback procedures) registered for this Hook will be called. In Apache 1.3, the hooks were defined by the module API, while Apache 2 allows adding new hooks. 37

- HTML** Hyper Text Mark-up Language is a document format which is used for documents in the World Wide Web. An HTML document basically contains the text, with formatting hints, information about the text and references to further components of the document like images that should be included when displaying the page. A web server's task is to deliver the HTML page and supply the resources referenced within the HTML page when requested. HTML documents can be static files or created dynamically by the server. 4
- Job Queue Server Model** A Multitasking Server Model using a job queue to provide communication between dedicated listener task(s) and a pool of idle worker tasks. A listener task waits for a request and puts a job entry holding the connection information into the queue. An idle worker task waits until it gets a job entry and uses the connection information to get and process the request.
An *idle worker queue* advises the listener to only accept requests if there is at least one idle worker in this queue. Else the request could not be processed immediately. 73
- Leader-Follower Server Model** A Multitasking Server Model using a pool of tasks changing their roles: One Listener Task is waiting for requests (the leader) while idle tasks (the followers) are waiting to become the new listener. After the listener task has received a request, it changes its role and becomes worker processing the request. One idle task now becomes the Listener. After the work has been done, the worker becomes an idle task. The Preforking Model (see p. 58) is a Leader-Follower Model 79
- Module (plug-in)** A separate piece of software intended to enhance an existing software system using a dedicated interface. This interface must be general enough to allow adding new functionality with future modules. An Apache module can register handlers for hooks that will be called in well-defined situations, and it can access resources of Apache and use its core functionality by the Apache API. 35
- Multiprocessing or threading** Multitasking means the concurrent execution more than one program by the same machine. Each task executes a program and uses resources of the machine. For each task, the machine provides a virtual processor equipped with a program counter (PC), a stack, and access to memory and I/O resources. In Unix, the tasks are called processes which don't share resources while in Windows, the tasks are called threads which use the same memory and I/O handles. 55
- Mutex** MUTual EXclusion mechanism, a means for inter-task communication usually provided by the operating system. Concurrent tasks use a mutex to ensure that only one task can access a shared resource. Any other task trying to get the mutex will either be blocked (suspended) until the mutex is released, or its request will be rejected. .. 72
- Port** In TCP/IP, a Port is part of an address. Each node on the Internet can be identified by its IP address. Using TCP or UDP, a node can offer 65536 ports for each of these protocols on each IP address, and assign ports to services or a client applications. A TCP or UDP packet contains both origin and destination addresses including the ports, so the operating system can therefore identify the application or service as its receiver. . 9
- Preforking Server Model** A Multitasking Server Model using a pool of Unix Processes (see Leader-Follower above). In contrast to the straightforward solution where a master server process waits for a request and creates a child server process to handle an incoming request, the Apache master server process creates a bunch of processes that wait for the requests themselves. The master server is responsible for keeping the number of idle servers within a given limit. 61



- Process** In the Multitasking context, a process is a task with high protection and isolation against other processes. The memory areas of two processes are separated completely. Processes can be executed using different user IDs resulting in different access rights. Sub-processes share file handles with their parent process.56
- Semaphore** A mechanism for inter-task communication, provided by the operating system. A semaphore is similar to a mutex, but provides a counter instead of a binary state. 72
- Stateless Protocol** A protocol is stateless if there is no relation between subsequent request-response pairs. The server can handle each request uniquely and does not have to keep a session state for the client.23
- Thread** In the Multitasking context, a thread is a task with low protection and isolation against other threads because they share memory and file handles. Usually, a thread runs in the context of a process. A crashing thread will not only die but cause all other threads of the same context to die as well.75
- URL** A Uniform Resource locator is an address for a resource on the Internet. Resource in this context can be a document, a file or dynamic information. URL usually contains information about the protocol which has to be used to access the resource, the address of the node the resource resides on and the location of the resource within the target node. Furthermore it can contain additional information like parameters especially for dynamically generated information. 11

Index

- .htaccess, 31, 85, 94, 97
- abstraction, 128
- accept mutex, 71, 72, 79
- accept(), 72
- access.conf, 33, 86
- agent, 123
- alarm(), 106
- ap_cfg_get_line(), 92
- ap_handle_command(), 92
- ap_process_resource_config(), 90
- ap_read_config(), 90
- ap_srm_command_loop(), 92
- Apache API, 49
- Apache core, 35
- Apache Portable Runtime, 49
- Apache Portable Runtime (APR), 51
- APR, 49, 51
- array management, 50
- attributes of an entity, 127
- Authentication
 - Basic, 20
 - Digest, 20
- authorization check, 84
- Basic Authentication, 20
- behavior, 123
- block diagram, 122
- brigades, 40
- browser, 4
- buckets, 40
- cache, 17
- cardinality, 127
- causal relationship, 125
- certificate, 21
- CGI, 24
- channel, 123
 - request/response, 123
- child server, 56, 61
- client, 4
- command table, 39
- command_handler, 89
- component, 123
- composition, 123
- compositional structure, 122, 128
- concurrency, 125
- configuration, 29, 65, 84, 85
 - global, 30, 85
 - local, 31, 85
 - per-request, 87
 - per-server, 86
 - syntax, 31
- Configuration Directive
 - Command Line, 90
 - Directory, 89, 92
 - Files, 89, 92
 - Location, 89, 92
 - Virtual Host, 94
- configuration directive, 30, 39
- conflict pattern, 125
- Connection Filters, 41
- connection socket, 109
- connection_rec, 50
- content handler, 39
- content negotiation, 15
- Content Set Filters, 41
- Cookies, 23
- core module, 89
- detach, 64
- diagram
 - block, 122
 - entity relationship, 126
- dialup socket, 109
- Digest Authentication, 20
- directory walk, 33, 94
- DNS, 9
- Domain Name Service (DNS), 9
- dynamic structures, 123
- entitiy
 - partitioning, 127
- entity, 127



- attributes, 127
 - relations between, 127
- entity relationship diagram, 126
- exec(), 104
- exponential mode, 70
- file walk, 97
- filter, 40
- filter chain, 40
- Filters
 - Connection, 41
 - Content Set, 41
 - Network, 41
 - Protocol, 41
 - Resource, 41
 - Transcode, 41
- firing predicate, 125
- firing rule, 125
- firing transition, 125
- FMC, 2, 121
- fork(), 57, 71, 103
- Fundamental Modeling Concepts, 2, 121
- gatekeeper, 56, 77
- global configuration, 30, 85
- graceful restart, 65, 69, 71
- handler, 37
- handlers, 35
- hierarchy of models, 128
- hook, 37
- HTTP, 11
- HTTP Method, 13
- HTTP server, 4
- httpd.conf, 33, 86
- HTTPS, 20, 22, 40
- idle queue, 79
- idle server maintenance, 69, 70
- idle_spawn_rate, 70
- Includes
 - Server-Side, 25
- INETD, 58
- informational system, 122
- initialization
 - child server, 62
 - first-time, 62
 - restart, 62
- invoke_command(), 92
- IP, 8
- iterative name lookup, 10
- job queue, 79, 80
- keep-alive, 16
- kill(), 105, 106
- Layer Diagram, 90
- Leader MPM, 79
- Leader-Follower, 72, 79
- listen socket, 109
- local configuration, 31, 85
- location, 123
- location walk, 98
- lookup_defaults, 89, 94
- Loop
 - Keep-Alive, 62
 - Master Server, 62, 65
 - Request-Response, 5, 62, 80
 - Restart, 61, 64
- loop pattern, 125
- master server, 56, 61
- master thread, 77
- mod_so, 87
- mode
 - exponential, 70
 - one process, 62, 64, 65
- model, 2
 - hierarchy of, 128
- module, 35
- module info, 35
- MPM, 74
 - Leader, 79
 - Per-Child, 80
- Multi Processing Modules, 74
- multiplex, 102
- Multitasking, 102
- name lookup, 10
- Network Filters, 41
- objectification of relationships, 127
- operation, 125
- other child, 69
- partitioning entities, 127
- pattern, 125
 - conflict, 125
 - loop, 125
 - sequence of actions, 125

- Per-Child MPM, 80
- per-request configuration, 87
- per-server configuration, 31, 86, 89
- per-directory configuration, 32
- per_dir_config, 94
- per_dir_defaults, 97
- per_directory_config, 87
- per_server_config, 87
- persistent connection, 16
- petri net, 123
- pipe, 110
- pipe of death, 60, 67
- place, 125
 - input, 125
 - marked, 125
 - output, 125
- pod, 67
- pool, 49
- port, 9, 57
- process, 102
- process_command_config(), 90
- protocol
 - stateless, 58
- Protocol Filters, 41
- proxy, 16
- raise(), 106
- REALMAIN(), 62
- recursive name lookup, 10
- refinement, 125, 128
- relations between entities, 127
- relationship, 127
 - causal, 125
- request, 4, 5
- request URI, 11
- request/response channel, 123
- request_rec, 50, 94
- resource, 4, 5
- Resource Filters, 41
- response, 4, 5
- restart, 65
 - graceful, 65, 67, 69, 71
- Restart Loop, 64
- scheduling, 102
- scoreboard, 65, 71
- sec, 89, 94
- sec_url, 89, 94
- sequences of actions pattern, 125
- server socket, 109
- Server-Side Includes, 25
- server_conf, 89
- server_rec, 50, 89, 94
- session information, 23
- set of entities, 127
- SIGHUP, 66
- signal handler, 105
- Signal mask, 106
- Signal set, 106
- signal(), 106
- Signals, 105
 - block, 106
 - ignore, 105
 - set of, 106
- SIGTERM, 66
- SIGUSR1, 66
- socket
 - connection, 109
 - dialup, 109
 - listen, 109
 - server, 109
- srm.conf, 33, 86
- SSI, 25
- SSL, 22
- standalone_main(), 64, 69
- stateless protocol, 58
- status
 - busy write, 82
- storage, 123
- structure
 - compositional, 122, 128
 - high-level, 128
 - lower-level, 130
 - value, 128
 - value range, 126
- supervisor process, 75
- synchronization, 125
- system
 - informational, 122
- table management, 50
- task, 102
- TCP, 8
- TCP connections, 8
- TCP/IP, 7
- TCP/IP connection, 5
- thread, 102
- Transcode Filters, 41



transition, 125

type of entities, 127

UDP, 8

Uniform Resource Identifier (URI), 11

URI, 11

URI translation, 94

value range structure, 126

value structure, 128

virtual host, 87

Virtual Hosts, 31

virtual hosts, 15

wait(), 69, 105

waitpid(), 105

walk procedures, 94

worker process, 75

worker thread, 77

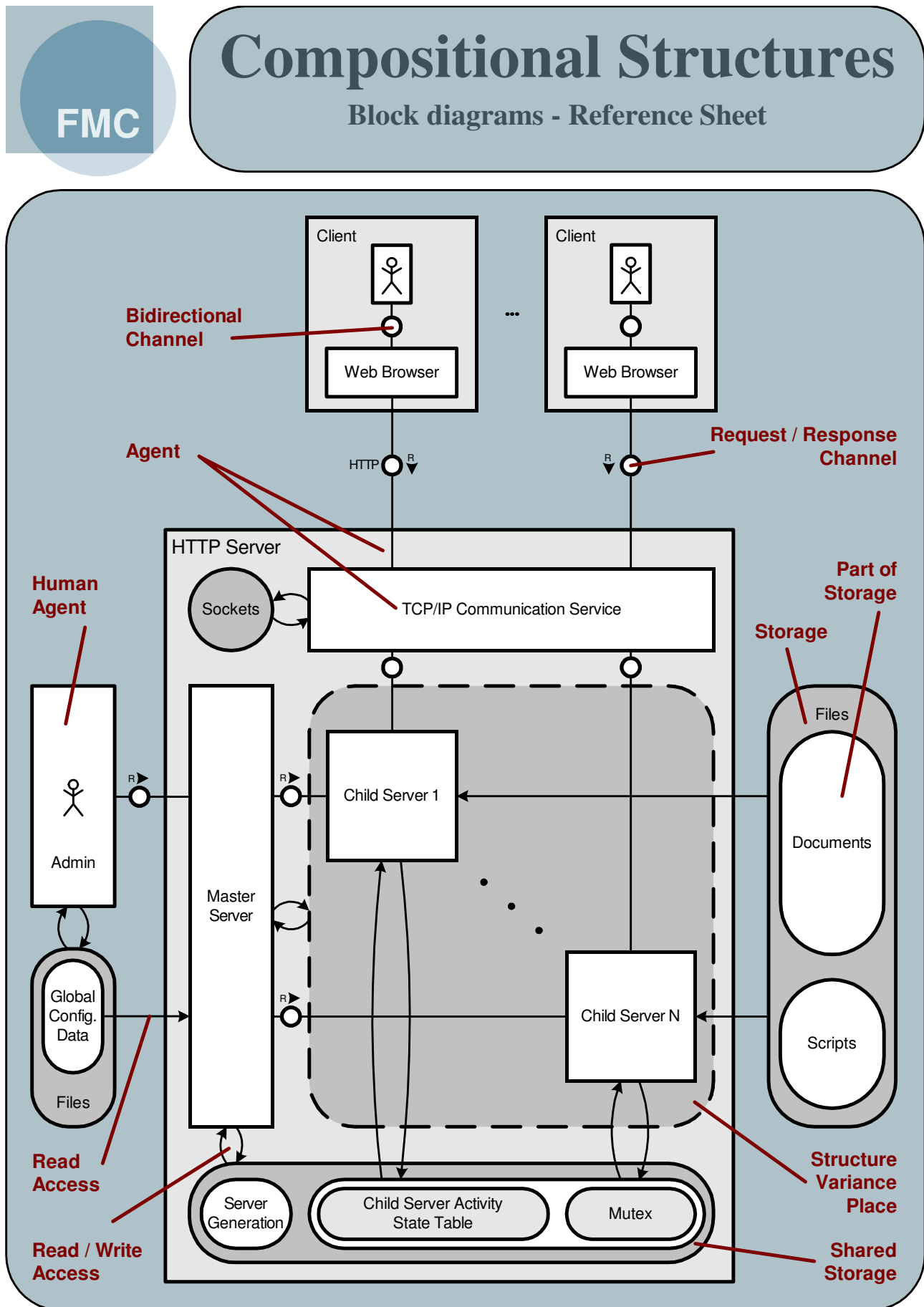


Figure D.1: FMC Block Diagrams (Part 1: Example)



FMC Block diagrams show the compositional structures as a composition of collaborating system components.

There are active system components called agents and passive system components called locations. Each agent processes information and thus serves a well-defined purpose. Therefore an agent stores information in storages and communicates via channels or shared storages with other agents. Channels and storages are (virtual) locations where information can be observed.

basic elements

	active system component : agent, human agent	Serves a well-defined purpose and therefore has access to adjacent passive system components and only those may be connected to it. A human agent is an active system component exactly like an agent but the only difference that it depicts a human. (Note 1: nouns should be used for identifier "A" Note 2: do not need to be depicted as rectangle or square but has to be angular)
	passive system component (location) : storage, channel	A storage is used by agents to store data. (Note: do not need to be depicted as ellipse or circle but has to be rounded) A channel is used for communication purposes between at least two active system components. (Note: channels are usually depicted as smaller circles but may also vary like the graphical representation of storage places)
	unidirectional connection	Depicts the data flow direction between an active and a passive system component.
	bidirectional connection	Like unidirectional connection but data flow is not strictly from one component to another one. Its direction is unspecified.

common structures

	read access	Agent A has read access to storage S.
	write access	Agent A has write access to storage S. In case of writing all information stored in S is overwritten.
	read / write access (modifying access)	Agent A has modifying access to storage S. That means that some particular information of S can be changed.
	unidirectional communication channel	Information can only be passed from agent A1 to agent A2.
	bidirectional communication channel	Information can be exchanged in both directions (from agent A1 to agent A2 and vice versa).
	request / response communication channel (detailed and abbreviation)	Agent A1 can request information from agent A2 which in turn responds (e.g. function calls or http request/responses). Because it is very common, the lower figure shows an abbreviation of the request/response channel.
	shared storage	Agent A1 and agent A2 can communicate via the shared storage S much like bidirectional communication channels.

advanced

	structure variance	Structure variance deals with the creation and disappearance of system components. An agent (A1) changes the system structure (creation/deletion of A2) at a location depicted as dotted storage. System structure change is depicted as modifying access. After creation agent A1 can communicate with agent A2 or vice versa.
--	--------------------	---

Figure D.2: FMC Block Diagrams (Part 2: Description)

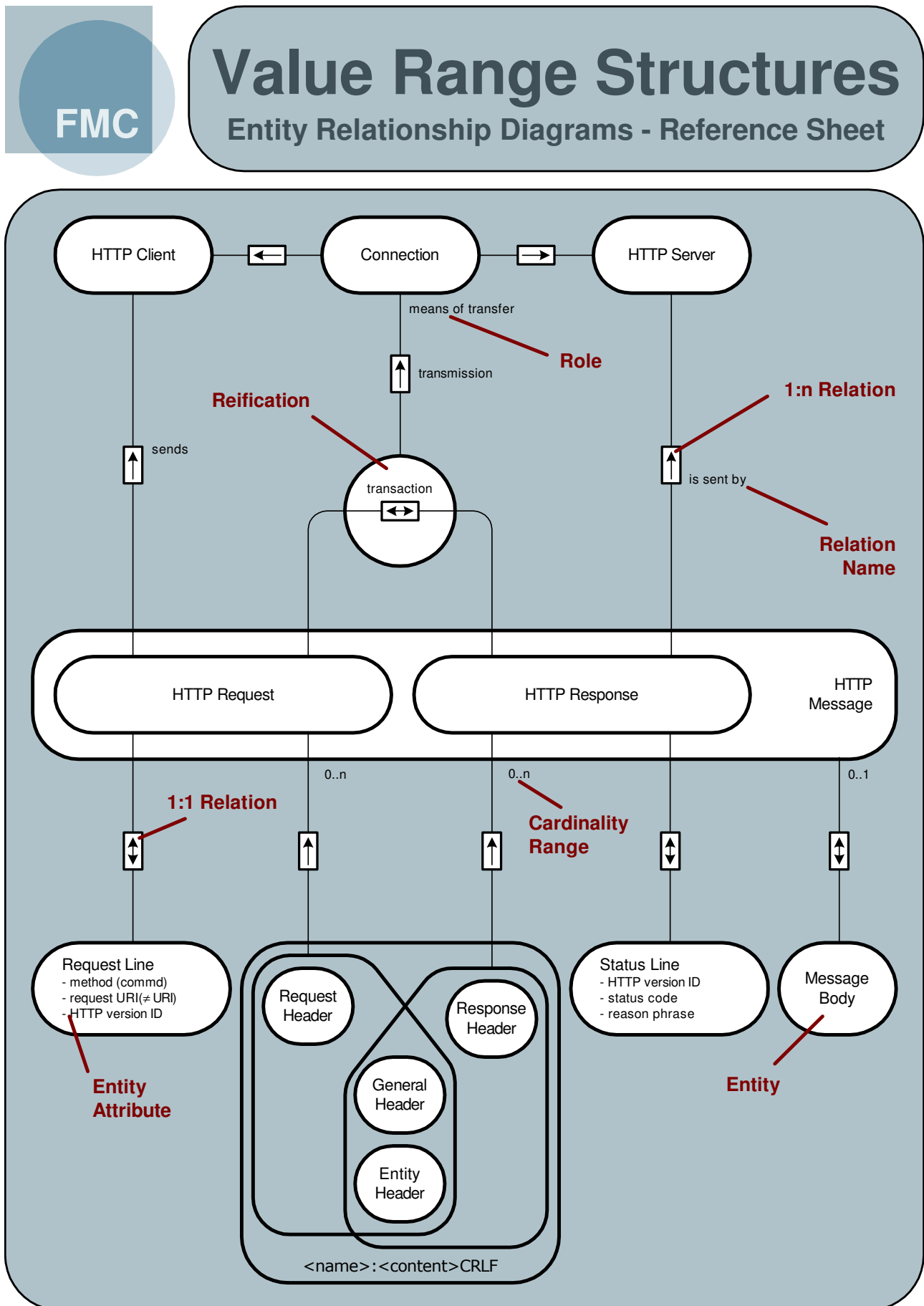


Figure D.3: FMC Entity/Relationships Diagrams (Part 1: Example)

FMC Entity Relationship Diagrams are used to depict value range structures or topics as mathematical structures.

Value range structures describe observable value structures at locations within the system whereas topic diagrams allow a much wider usage in order to cover all correlations between interesting points.

basic elements		
	entity / entity set	Entities or sets of entities participate in relations. Furthermore some attributes (A1 ... An) might be specified. (Note: singular nouns should be used for identifier "E")
	relation (n:m, 1:n, 1:1)	Is a subset of the cross product of all participating entities, i.e. they relate entities to each other. If the relation identifier "R" is aligned with one of the entities (usually verbs are used in this case) it should be read from this direction. If the relation identifier "R" is aligned in the middle of the relation there is no direction (usually nouns are used in this case).
	arc	Connects a relation and an entity. A cardinality range may specify how often the respective entity takes part at least and at most in the relation similar to the (min,max) notation. Furthermore a role might clarify the kind of participation of the entity, which is especially useful for recursive relations. (Note: singular nouns should be used for identifier "role")
further elements		
	orthogonal partitioning	Additional partitioning of an entity which is independent from any previous partitioning.
	structure entity	Is used to create an entity from a structure (entities and relations).
common structures		
	n:m relation	Each element of E1 occurs i to n times in the relation with E2 while each element of E2 occurs j to m times in the relation.
	1:n relation	Is like an unique function $f(x \in E1) = y \in E2$. Each element of E1 is associated with exactly one element of E2. (Note: the cardinality ranges in the parentheses should be assumed due to the arrow symbol inside the relation. Deviant cardinality ranges must be mentioned explicitly.)
	1:1 relation	Is like an one-to-one function. One element of E1 is associated to exactly one element of E2 and vice versa. (Note: the cardinality ranges in the parentheses should be assumed due to the arrow symbol inside the relation. Deviant cardinality ranges must be mentioned explicitly.)
advanced		
1) n ary relation (e.g., ternary)	2) reification	3) orthogonal partitioning
<p>1) Sometimes it is necessary to correlate more than two entities to each other via n ary relations. The example shows a ternary relation.</p> <p>2) Elements of a relation constitute the elements of a new entity, which in turn can participate in other relations. The example shows the relation C being reificated.</p> <p>3) Partitioning of entity E into the entities X, Y and additional, independent partitioning of entity E into the entities A, B. Imagine for instance entity E as "Human Being". Then entity X may stand for "Man", Y for "Woman" and thereof independently entities A and B could mean "European" and "Non-European".</p>		

Figure D.4: FMC Entity/Relationships Diagrams (Part 2: Description)

Dynamic Structures

Petri nets (1/2) - Basic Reference Sheet

FMC

FMC diagrams for dynamic structures are based on transition-place Petri nets. They are used to express system behaviour over time depicting causal dependencies. So they clarify how a system is working and how communication takes place between different agents.

Here only the basic notational elements are covered whereas the rest is located on another - more advanced - reference sheet (2/2).

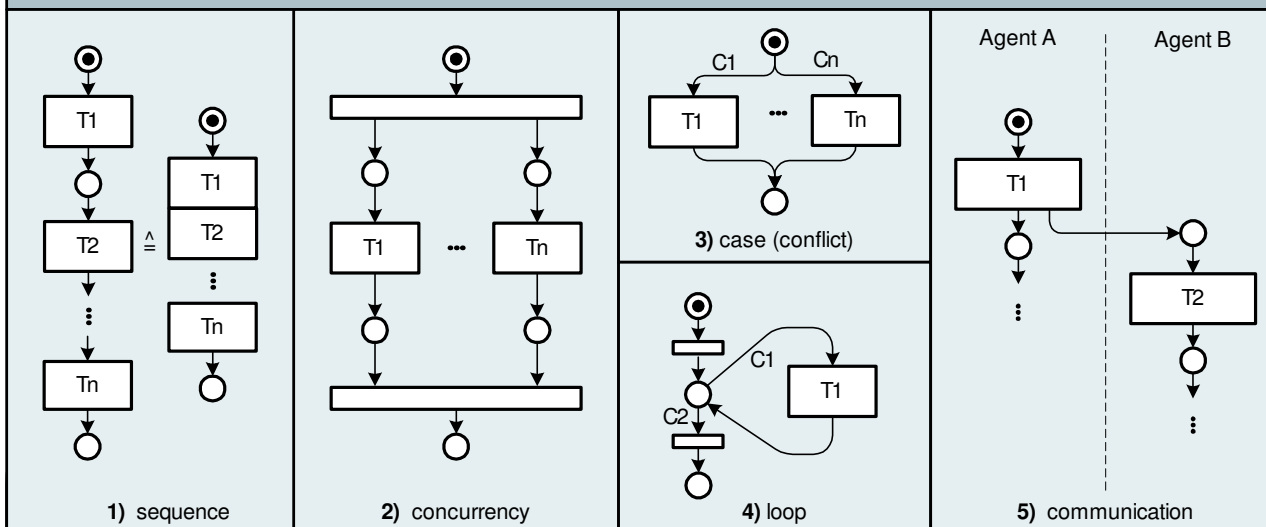
basic elements

	transition	Stands for an operation, an event or an activity. (Note: verb should be used for identifier "T")
	place	Represents a control state or an additional condition. (Note: capacity = 1)
	directed arc	Connects a place and a transition.

further elements

	NOP	A transition meaning No OPERATION. (Note: often used to keep the bipartiteness)
	swimlane divider	Distinguishes competences of agents.

common structures



- 1) Defines that transition T1 fires first, followed by transition T2, followed by transition T3 ...
- 2) Means that transitions have no causal ordering. The transitions T1, ..., Tn are concurrent, the firing of T1, ..., Tn has no special order.
- 3) Is used to choose one transition among others. Only one of the transitions T1, ..., Tn will fire, depending on the conditions C1, ..., Cn associated to the arcs.
- 4) Is used to repeat the firing. Transition T1 will be repeated as long as condition C1 is fulfilled. Often C2 is not mentioned as it is assumed to be "else".
- 5) Whenever a swimlane divider is crossed communication takes place. Upon this structure all possible communication types can be expressed (synchronous, asynchronous etc.).

Figure D.5: FMC Petri Nets (Part 1: Basic components)

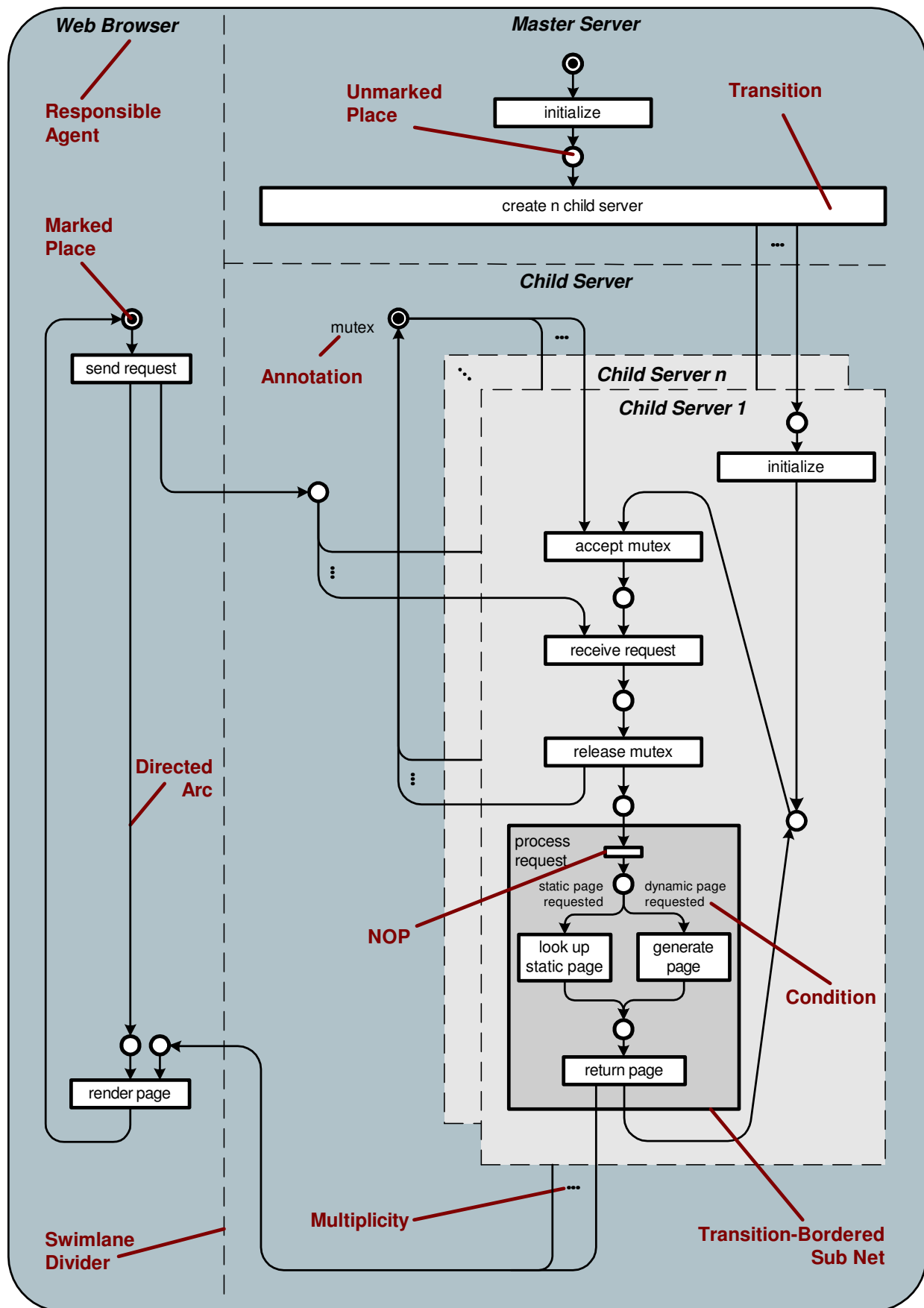


Figure D.6: FMC Petri Nets (Part 2: Basic Example)

Dynamic Structures




Petri nets (2/2) - Advanced Reference Sheet

FMC



FMC diagrams for dynamic structures are based on transition-place Petri nets. They are used to express system behaviour over time depicting causal dependencies. So they clarify how a system is working and how communication takes place between different agents.

Here only the advanced notational elements are covered whereas the rest is located on the basic reference sheet (1/2).

extended elements

 	multi-token place	Places which can hold multiple tokens but not an infinite number are indicated as enlarged places with an annotation specifying the capacity ($n > 1$). Places with an infinite capacity are indicated by a double circle.
	arc	The arc weight n determines how much tokens will be consumed or produced when the connected transition fires. An arc weight of 1 is assumed, if there is no one specified.

recursion elements

	stack place (cap. 1, cap. infinite)	Is a place to store information about return positions using stack tokens. All stack places with the same name are strongly coupled with each other as the stack tokens, although placed on several stack places, are managed in a single stack. So all the stack places together constitute the return stack.
	return place	Is used like a normal place. But there is always a conflict to solve as a return place is an input place for at least two transitions that also have stack places as input places. When a return place gets a token and more than one associated stack places have a stack token the conflict is always solved in the same manner: the newest token on the stack must be consumed first. The newest token belongs to exactly one stack place and so the transition where this stack place is an input place will fire.

general recursion scheme

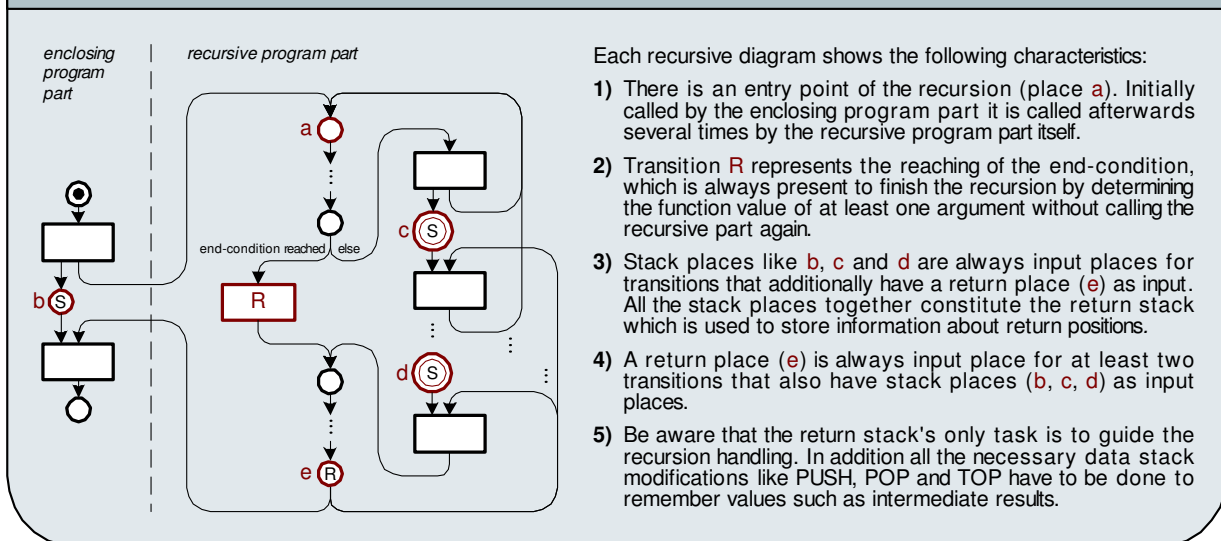


Figure D.7: FMC Petri Nets (Part 3: Advanced features)

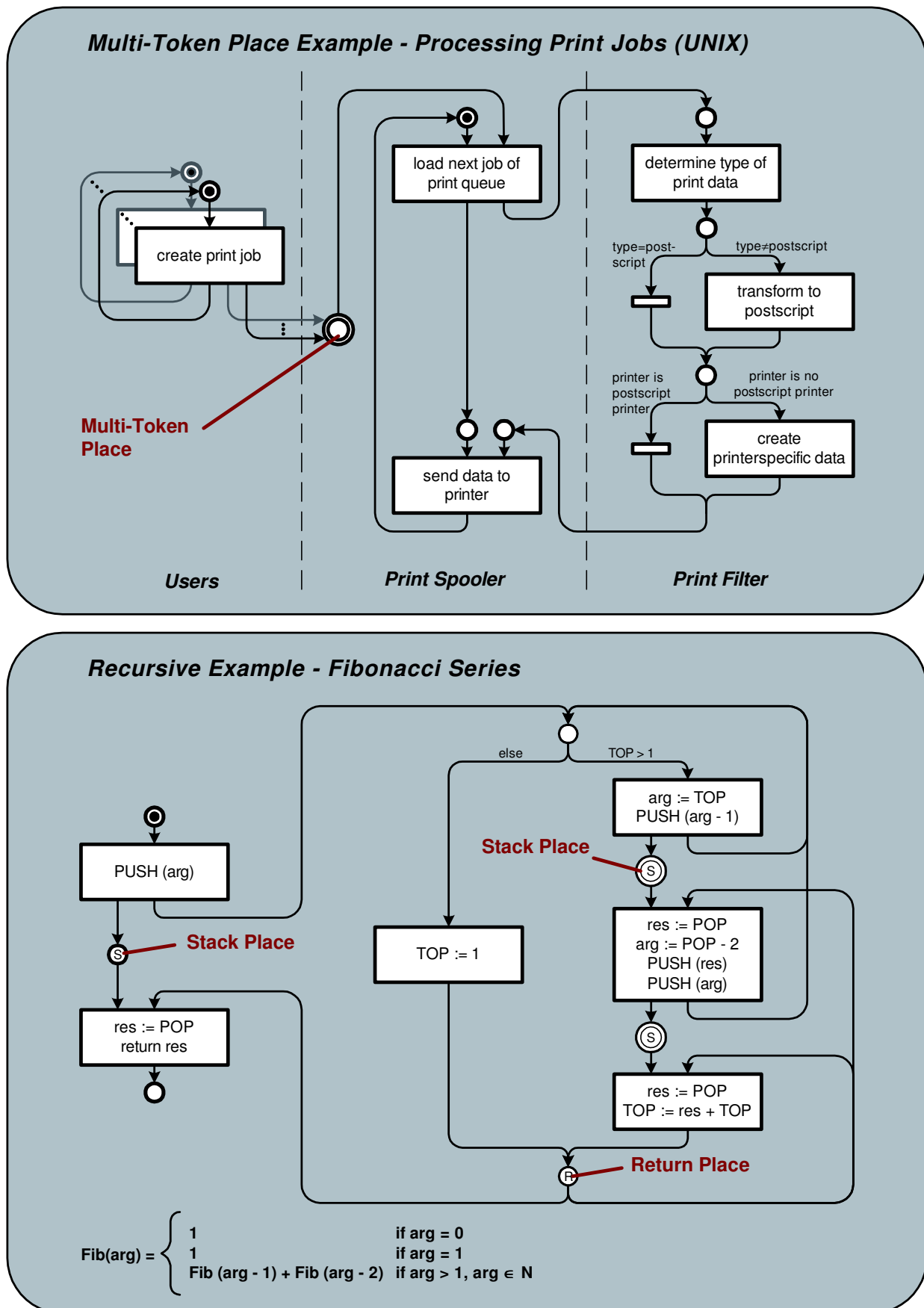


Figure D.8: FMC Petri Nets (Part 4: Advanced Example)

ISBN 9-937786-14-7
ISSN 1613-5652