ISSN 0280-5316
ISRN LUTFD2/TFRT--5796--SE

Preparing the Apache HTTP Server for Feedback Control Application

Erik Lindegren

Department of Automatic Control
Lund University
February 2008

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
February 2008

Document Number
ISRN LUTFD2/TFRT--5796--SE

Author(s)
Erik Lindegren

Supervisor
Anders Robertsson and Martin Kjaer Automatic Control, Lund
Björn Wittenmark at Automatic Control, Lund (Examiner)

Sponsoring organization

Title and subtitle
Preparing the Apache HTTP Server for Feedback Control Application (Användning av reglerteknik I Apache HTTP-servern)

Abstract
In the last couple of years it has become more and more common to use control theory in computing systems, for instance operating systems, web servers and databases. The reason for this is to make these systems more robust and stable. This is of interest because we today are more dependent on computing systems in our everyday life, and therefore put higher demands on them. A lot of focus has been on internet service systems and this paper also deals with such a system, namely the HTTP server.
In contrast to mechanical systems where you often have a good intuition of where the problems occur, it might be hard to get the same overview of a computing system. Therefore an introduction to HTTP servers, and the Apache HTTP server in particular, is presented in this thesis focusing on the problems which may arise in such a system.
When the important issues have been identified a solution of how to apply control theory to the Apache HTTP server is presented. Changes are made to the Apache HTTP server which make relevant measurements of the server's performance available during runtime and make it possible to influence the server's behaviour by updating important configuration parameters in realtime. A simple controller is then implemented using this new functionality.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
107

Recipient's notes

Security classification

Contents

Chapter 1

Introduction

Today when the use of computer systems is higher than ever, the requirements on these systems have also increased. In particular, we have become more dependent on the Internet and the various services it provides. The Internet is used for, among other things, managing bank accounts, booking holidays and reading the news. According to Statistics Sweden [1] more than 80% of all people in Sweden aged 16 to 74 had access to the Internet in their homes during 2007. It is therefore important that we can rely on these system. This is especially true during a crisis. As example one can mention the September 11 attack in New York 2001, the murder of the Swedish foreign minister Anna Lindh 2003 and the tsunami crisis 2004. At the time of these events, the demand for information was so high that some systems became overloaded and therefore not available.

What people try to achieve by using control theory in computing system is to make them more robust and stable. The most popular areas of research have been data networks, operating systems, middleware (e.g. web servers, database servers), multimedia and power management [2].

Because of my background as a web developer my main interest in this development has been in web servers. In this thesis I am going to examine a web server and see what different areas of it that would be good to have under control and what is required of the web server to make this possible. First take a look at a very simplified model of a web server.

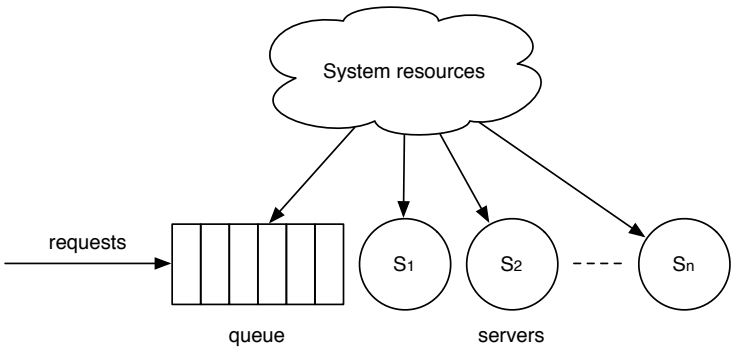

Figure 1.1: Simple web server model.

It has a queue for incoming client requests and a number of servers ready to serve these requests. The length of the queue is limited and there is also a timeout that defines how long a request can stay within the queue before it is removed without being processed. The computer, on which the web server is running, has different system resources that the web server needs to use. These include CPU time, main memory (RAM), processes, TCP connections, TCP buffers and file system buffers [3]. In this thesis when talking about the load on the server I refer to the load on the computer, i.e. the usage of its system resources.

What different types of control would be interesting to apply to such a system? To answer that question one need to think about what kind of problems that might occur within this system. Knowing what these issues are, one can start to think about how to avoid them by using control. So what might these issues be?

1.1 Overload

As soon as there is a queue in a system with a fixed length there will always be a possibility that this queue will fill up. When the queue becomes full the system will not be able to accept any more requests from the clients, which will instead be blocked. One of the reasons for having a fixed queue length is to avoid overloading the server. Allowing every client's request into the server during a busy period might lead to a very large number of requests in the queue. Because every request will consume system resources that are also needed by the servers, this can cause the system to become overloaded. When this happens the servers might not be able to serve a request particularly fast or will not be able to serve the request at all which in the end might lead to a system crash.

Overloading the system can cause major problems and is something that should be avoided in every situation. There are a number of possibilities on how to solve this. To have a fixed queue length is one way of doing it, but it is not enough. The load each client's request has on the server may vary a lot. One request might be very complex and need a lot of system resources while another one might not need much at all. This makes it very hard to set a fixed queue length that will guarantee that the system never becomes overloaded. Therefore it could be a good idea to let a controller regulate the queue length depending on the server load. It could also be beneficial if it was possible to limit the number of requests in the queue by controlling a timeout value, which tells a request how long it can stay in the queue before being removed.

But to only control the number of requests in the queue will not be a guarantee for avoiding a server overload. There will still be a limit on the system resources which need to be divided between all parts of the server. As mentioned earlier there might be huge differences in system resource usage depending on the client's request. Therefore it would probably also be of interest to be able to control the number of servers running depending on the load.

Web server overload control is investigated in [4] with focus on overload control techniques such as admission control and content adaptation (see Section 3.5 on page 31).

1.2 Quality of Service

Often when talking about a server system it is important to think about the quality of service (QoS) that a system offers to the client. There are various properties that can be observed to measure this, but the most important ones for web servers are the following:

- *Response time* which is the total time it takes for the system to respond to a client's request.
- *Throughput* which is the rate at which the system can handle requests.
- *Availability* which is the percentage of time that the system is available

Depending on what quality you have chosen to be the most critical one for your system, you can measure these variables to get an indication on how well your system is performing.

1.2.1 Availability

Availability is closely linked to the queue. There are three possibilities that will make the system un-available:

1. During an overload situation
2. When the queue is full the system will not accept any more client requests and therefore not be available.
3. If the server is very busy a request in the queue might timeout before it has been served

To control the system availability you should therefore regulate the queue length, the queue timeout and avoid overload.

1.2.2 Throughput

To improve the throughput of the system the number of servers could be increased. This will make it possible for the system to handle more requests simultaneously. One could also increase the queue length and the queue timeout. This will let more requests into the system and therefore decrease the idle time of each server and in turn improve throughput. But it is important to be careful when changing these values, because an increase in any of these parameters will consume more system resources. If the usage of system resources becomes too high it will lead to server overload and a dramatical decrease in throughput.

1.2.3 Response time

It is a bit more difficult to say what would improve the response time of the system because it is highly dependent on the server load. One setting that will give really good response times during light load might not work that well under heavier load. It is desirable to keep the time each request spend in the queue as low as possible.

The queue is a good thing to have for both the availability and the throughput, because it makes it possible for the system to accept more requests even though all the servers might be busy at a particularly point. This is not true at all for the response time. A queue is not something that improves response time. To get really good response times a request should not have to stay in the queue at all, it should be processed immediately. So if the response time is the most critical parameter for our system we might reason that it is better to keep the queue length and queue timeout as low as possible and risk that a request might be blocked. In that case it could be better to block requests if they can not be served immediately. This should lead to a scenario where the requests that are handled will be handled quite quickly. To have a lot of servers handling requests simultaneously will probably increase the response time, because this will consume more of the system resources, which are limited. Therefore the number of servers should be kept quite low.

To make the system perform well in all of these three parameters is difficult because there are conflicts between these parameters. An improvement in response time performance might lead to a decrease in both throughput and availability. Throughput and availability go hand in hand. An improved throughput will probably also be good for availability but will increase the response time.

1.3 Platform

From this discussion it is possible to identify server directives that would be good to be able to update during runtime in order to improve the system's performance: queue length, queue timeout and the number of servers. Is there a web server that allow updates to these kinds of directives during runtime?

The four most popular web servers (at the moment this thesis was written) are the following [5]:

- Apache HTTP Server from the Apache Software Foundation [6]
- Internet Information Services (IIS) from Microsoft
- Sun Java System Web Server from Sun Microsystems, formerly Sun ONE Web Server, iPlanet Web Server, and Netscape Enterprise Server
- lighttpd [7]

None of these web servers have support for this, so what needs to be done is to rewrite one of them to be able to facilitate this functionality. Both the Apache HTTP Server and lighttpd are free open source projects, which makes it possible for me to actually alter the server code. Of these two I chose to work with the Apache HTTP Server because it is the most popular one.

As operating system Linux was chosen because it is an open source project and because it is the most common platform for running the Apache HTTP Server. Therefore all the system descriptions in this thesis are related to that platform.

1.4 Thesis Outline

In Chapter 2 the background and information related to the Apache web server will first be described. In Chapter 3, I will present details of how to make it possible to influence the server's performance during runtime. In Chapter 4, I will explain how to measure the server's performance. In Chapter 5, the implementations that were made in the previous two chapters will be tested by running some experiments on the Apache HTTP server. In Chapter 6, I will briefly discuss where a controller should be placed and how it could be used. In Chaper 7, the results will be summarized and suggestions of further work presented.

1.5 Related Work

The largest inspiration for this thesis has been the book *Feedback Control of Computing Systems* by Joseph L. Hellerstein, Yixin Diao, Sujay Parekh and Dawn M. Tilbury [2]. It is a good start if you are interested in using feedback control in a computing system. I particularly liked the fact that many of the examples used real life computing systems like the Apache HTTP server and the IBM Lotus Domino Server. Another resource written by the same authors that has been important for this work can be found in [8].

In addition to this the department of Automatic Control in collaboration with the department of Communication System at Lund Institute of Technology have done interesting work about using admission control for web server systems, which also has also been an inspiration for my work, see for instance [9] [10] [4].

Chapter 2

Background

2.1 Web Server Basics

A web server is a server system on the internet which is responsible for handling HTTP requests from HTTP clients (e.g. browsers). A more correct definition would therefore be an HTTP server. HTTP stands for Hyper-Text-Transfer-Protocol and specifies how requests to the server and responses from the server should look like. The communication between a client and a server always starts with a request from the client. The first line in a HTTP request contains information about what function the client want to use (e.g. GET, HEAD, POST), a Uniform Resource Identifier (URI) telling the server on what resource the client want to use this function and finally information about the HTTP version. The rest of the lines contains the headers, which are used to add information about the request or modify it. Here is an example of how a simple HTTP request may look like:

```
GET /index.html HTTP/1.1
Host: www.epineer.se
```

From this request the server knows that it should get the index.html file for the host www.epineer.se and return its content to the client. The response from the server looks like this:

```
HTTP/1.1 200 OK
Date: Fri, 19 Jan 2007 23:06:47 GMT
Server: Apache/2.0.52 (Red Hat)
Last-Modified: Wed, 15 Feb 2006 16:50:21 GMT
ETag: "1640c0-32-1b84cd40"
Accept-Ranges: bytes
Content-Length: 50
Content-Type: text/html

<html>
<body>
<h1>epineer.se</h1>
</body>
</html>
```

The first row of the response contains the HTTP version and the status of the request (in this case 200 which means that the server managed to handle the request successfully). The next two

rows contain information about the date and various data about the server. The rest of the lines are the response headers. The content of the index.html file is located after the headers separated by an empty line.

When a client has received a response for its request the HTTP communication between the client and server stops until the client sends a new request. This makes HTTP a stateless protocol, because it does not use a wait time before closing the communication which is quite common in other protocols.

2.2 The Apache HTTP Server

2.2.1 Introduction

To be able to control the Apache HTTP Server the first step is to get a general idea of how it works. This is going to be described in this section. To get a more complete understanding of the server I can recommend the excellent document "The Apache Modeling Project" [11]. From reading the source code the authors have described how a large number of different areas of the server are implemented. It is a good reference to have when reading the source code. The home page for the server [6] is also a helpful resource. All of the Apache HTTP Server's code is written in the programming language *C*, so a good *C* programming book [12] might come in handy. The GNU *C* library is used as the standard *C* library on most systems running the Linux kernel, so its reference manual, which can be reached from [13], may also be of use.

2.2.2 History

Since 1995 the Apache HTTP server, commonly referred to as Apache, has been the most popular web server according to the netcraft survey [5]. The first version of Apache was released in 1994 and was based on the NSCA HTTP server. It was called the NSCA server because it was developed at the National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign by Rob McCool. At the top of the main configuration file for Apache called *httpd.conf* it still reads:

> Based upon the NCSA server configuration files originally by Rob McCool.

In 1994 they stopped working on the NSCA server and that is why the Apache group was first created. Some of the webmasters that had used the NSCA server had written their own fixes (so called patches) for that server. They got together and gathered all the patches and released a more stable server which they called Apache. The name is said to be derived from "a-patchy-server".

2.2.3 Basic Architecture

The Apache HTTP Server is a multi-task server, which means that it can serve a lot of different requests simultaneously. How the multitasking is implemented is highly dependent on what operating system the server is running on. However, all of the different multitasking models for Apache share some basic features: All of them use a task-pool and a master server to control this pool. When starting the Apache server what is started is the master server. It then creates a number of tasks (that can be implemented as processes or threads or a combination of the two) which are responsible for handling incoming requests. The main responsibilities for the master server is to create and delete tasks in the pool. By using a command-line interface called *apachectl* you can send different signals to the master server. For instance when standing in the base folder of your Apache installation, to start and stop the server you type:

```
$ ./bin/apachectl start
$ ./bin/apachectl stop
```

It is through the master server that server administrators interact with Apache. The master server will run as long as no serious problem occurs or until the administrator decides to stop/restart the server.

To be able to make the Apache server runnable on a large variety of platforms in an elegant way, all the code for the multitasking model have been broken out and made into different loadable modules. These are called Multi-Processing Modules (MPM:s) and defines what strategy the master server will use to dispatch its tasks. When configuring and installing an Apache version choices are made regarding what MPM:s to use. Here are the default ones for a number of platforms:

BeOS	beos
Netware	mpm_netware
OS/2	mpmt_os2
Linux/Unix	prefork
Windows	mpm_winnt

I was running the Apache server on a Linux platform and chose to use the default Prefork MPM, which also was the first multitasking architecture of Apache.

2.2.4 The Prefork Multitasking Architecture

In the Prefork MPM the master server and all of the tasks are implemented as processes[1]. The master server, from now on called the parent process, reads the configuration files and creates the other processes in the pool by using the system call *fork*. To see the Linux manual page for this system call type the following in the command line:

```
$ man 2 fork
```

The *fork* system call creates a child process by making an exact copy of the parent process. By copying the parent process the child will get the same configuration. Like most other configuration parameters these are entered in the main configuration file *httpd.conf*. A typical configuration for Prefork might look like this:

StartServers	8
MinSpareServers	5
MaxSpareServers	20
MaxClients	256
ServerLimit	256
MaxRequestsPerChild	4000

The amount of child processes that will be started initially at server startup by the parent process is defined by the parameter *StartServers*. Prefork always tries to keep a certain number of idle child processes, to make it possible for a request to be served instantaneously without having to wait for a child process to be created. This is the reason why this MPM is called pre-fork, because it forks a child process before there are any requests to serve. The parameter *MaxSpareServers* tells Apache the maximum number of idle processes in the system and the parameter *MinSpareServers* the minimum number of idle processes. During operation Apache will try to keep the amount of idle servers between those two values. The maximum number of server processes in the system is defined by the parameter *MaxClients*. The *ServerLimit* is used to tell Apache the maximum number of child processes it should expect and use this value to instantiate enough entries in the scoreboard (see page 8). This means that the *ServerLimit* will define an upper

[1] A process is a program in execution and is defined by the resources it uses and by the location at which it is executing.

limit for the *MaxClients* value. The *MaxClients* value can be changed during runtime by using a graceful restart (see page 9), while the *ServerLimit* can not. To limit the number of requests that a child process can handle during its lifetime the *MaxRequestsPerChild* value is used. If this value is set to 0 it will mean an unlimited amount of requests. The reason for wanting to limit a child process lifetime is to be on the safe side if any child process would start consuming a lot of system resources, e.g. memory caused by a memory leak. Apache itself should be safe against memory leakage, but a third-party module that a child process uses may not.

The Scoreboard

The parent process needs a way of communicating and keeping track of all the child processes. This is done through something called the scoreboard. The scoreboard is a simple data structure (a *struct* as it is called in *C*) stored in a shared area (usually shared memory, but a file may also be used) that both the parent and the child process can access. It will therefore not be affected by the context switch. The scoreboard consists of the following three areas:

global A *struct* containing global parameters that both the parent and child processes should be able to access, e.g. the generation id (used for graceful restart, see page 9) and the *ServerLimit* parameter.

parent A *struct* containing various parameters that mostly the parent will use, e.g. the process id (PID) for each child process.

servers A table containing various information about the child processes like the status (e.g. dead, ready, starting), the start time of a request, the stop time of a request, how many bytes that was served for the request etc. The scoreboard has functions to update these parameters. When a child is created it will get a unique id that is used as an index into this table. By using this id, the parent is able to gather information about a specific child process.

In the source code, the instance of the scoreboard is called *ap_scoreboard_image*.

Idle Server Maintenance

When the parent process enters its main loop, it does nothing except waiting for a child process to die. If this happens or if the waiting time exceeds a timeout of one second the function *perform_idle_server_maintenance* will be run. It is in this function that the parent controls its pool of child processes, by determining how many children to spawn or to kill. The function starts by looping through all the child processes already in the scoreboard and examines their status. Each process with the status *dead* will get its slot in the scoreboard marked as free. As soon as the number of free slots is equal to the amount of children it wants to create the loop stops running. If there is not enough old slots with the status *dead*, the function will increase the number of slots in use until there is enough, just as long as the number of slots in use is below the *MaxClients* value. When the function loops through the child process it also registers how many of the servers that were idle. If the number of idle servers exceeds the parameter *MaxSpareServers* one of these child processes will be terminated. If the number of idle servers is less than the parameter *MinSpareServer* the spawn rate is increased, and the next time the function runs those extra processes will be created.

The Pipe-Of-Death

In *C* programming pipes may be used as a simple way for different processes to communicate with each other [13]. In the server the parent and all the child processes share a pipe that are called *pod* in the source code. The *pod* stands for "Pipe-Of-Death" because it is used by the parent to kill child processes. The *pod* has just enough space to contain one character, and the processes

may read or update this character. To terminate a child process the parent may write a character called the *Character-Of-Death* to the pipe by calling the function *ap_mpm_pod_signal*. As soon as a child process has finished serving its request it reads the *pod* to decide whether it should continue listening for new requests or die. If it reads the *Character-Of-Death* it will remove the character from the *pod* and terminate.

Graceful restart

In the Apache HTTP Server it is possible to reconfigure the server during runtime by doing a graceful restart from the command line, e.g. if there is a need to change the *MaxClients* value without taking down the whole server. This value is first edited in the *httpd.conf* file and then the following command is issued to make the running server update its configuration:

```
$ ./bin/apachectl graceful
```

When choosing to do a graceful restart the parent process will first re-read the configuration files and loop through all of its child processes. The idle ones will be replaced with new processes with the updated configuration. The busy child processes will be replaced after they have finished serving their request. Each time the server is restarted it will get a new global generation number. This generation number is stored in the scoreboard with each child process. When the process generation number is different from the current global one, it knows that it should be replaced by a new child process.

Persistent Connections

In the beginning there was a problem with web servers if a web page used a lot of images or other forms of embedded objects. Each object would require a new request to the server, and therefore a new TCP connection to the server (see Section 2.3 on page 15). When there were a lot of different objects on a page, this would consume a lot of server power. To get around this, a support for persistent connections was added to the Apache server. This means that a request for a page and the requests for the embedded objects in the page will all share the same TCP connection. The parameters for configuring persistent connections in Apache are:

KeepAlive	On
KeepAliveTimeout	15
MaxKeepAliveRequests	100

KeepAlive simple tells Apache to use persistent connections or not, the default value is *On*. The *KeepAliveTimeout* parameter decides how many seconds a child process should wait for a new request from the same client. If the next request comes within this period the same connection is used to handle the new request, if not the persistent connection is dropped. When *KeepAlive* is used only the first request counts toward the *MaxRequestsPerChild* parameter in Prefork. Another changeable parameter is the *MaxKeepAliveRequests*, which tells Apache how many requests a single persistent connection is allowed to handle before it should close down the connection.

2.2.5 Code base

In this thesis the 2.0.54 source distribution of the Apache HTTP server is used. Figure 2.1 shows an overview of how the code is organized for the 2.0.45 distribution which has the identical code structure as 2.0.54.

For this project I have made code changes to the following files:

```
include/scoreboard.h
modules/http/http_request.c
```

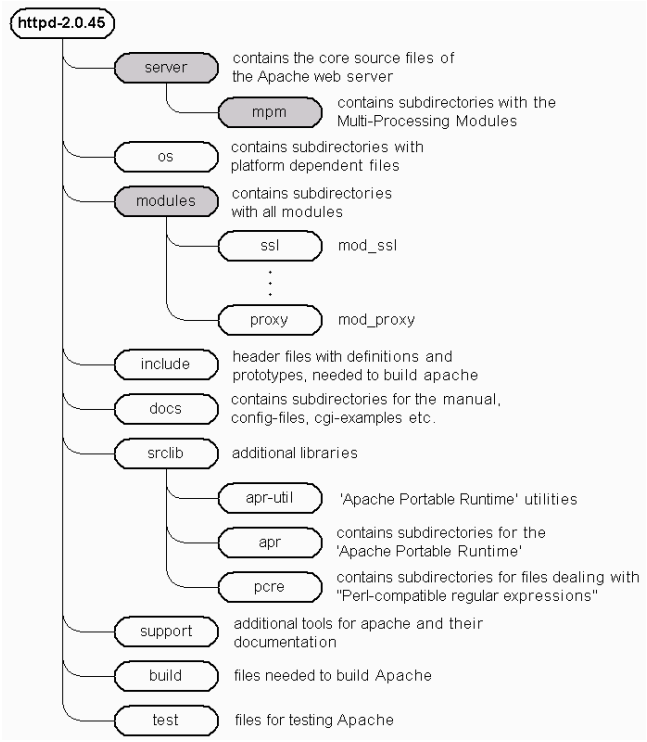

Figure 2.1: Directory structure of the Apache HTTP Server 2.0.45 source distribution [11]

`server/mpm/prefork/prefork.c`
`server/Makefile.in`
`server/scoreboard.c`

and added functionality for logging and control in:

`include/logger.h`
`server/controller.c`
`server/logger.c`

2.2.6 Apache Portable Runtime

To make the Apache HTTP server a truly portable project, i.e. make it possible to run it independently of the operating system, all the platform-specific implementations have been moved into a supporting library called Apache Portable Runtime (APR) [14]. It contains functions for doing a number of things, which means that the software developers do not have to worry so much about the platform. When they have written their code by using the APR-functions they can be pretty sure that it will work the same regardless of the underlying system. All the functions in the APR library have names that starts with *apr_*. Here are some examples of what functionality APR provides:

- Memory allocation and memory pool functionality
- File I/O
- Thread, process and mutex functionality
- Shared memory functionality

- Network sockets and protocols

The Apache Portable Runtime project started as a subproject to the Apache HTTP server, to add support to run the server on different platforms. It has now grown and become its own project, which can be used to create other platform-independent projects [15].

2.2.7 Modules

It is possible to extend the functionality of the server by using Apache modules [11]. They may be included statically or dynamically in the server. Which ones to include statically is chosen when configuring and installing the server. They are compiled into the server and can not be altered without rebuilding the whole server. What other modules to use is something you specify in the configuration file. These will then be dynamically loaded into the server during startup. The executable code of the server consists of the core, static modules and dynamic modules. The child processes will contain exactly the same executable code as the parent process because they are copies of the parent. Therefore all processes will contain all modules.

There are quite a lot of different Apache modules, and which ones to include in the server can be difficult to decide [16]. If no module with the desirable functionality exists, it is possible to write a new Apache module [17].

All the modules interact with the server through a common interface. Each module can register a variety of handlers in the server core:

Hooks A so called *Hook* is triggered when a certain event occurs e.g. server startup, server shutdown, child process creation, child process termination, etc. When a *Hook* is triggered Apache calls the handlers that the modules have registered for it.

Configuration directives It is possible for the modules to define their own set of configuration directives, which then can be used in the standard configuration file. At startup the server calls the corresponding handlers specified by the modules to process these directives.

Filters A multiple array of modules that work together can be used to manipulate data of a response or a request. These modules are called filters. For each content type that the server handles you can register multiple filters and also specify in what order these filters should be applied to the data.

Optional functions An optional function is very similar to a *Hook*, but with the difference that the server ignore any return value. So even though an error might occur, the server will call all the optional functions.

The way an Apache module interact with the server is through the Apache Application Programming Interface, a so called API. It contains functions which a module can use to perform various tasks. All the functions have names starting with *ap_*. Some of these functions will in turn be based on the APR-functions. The most important service the API provides for the modules is functions for memory management. The Apache server uses a pool based memory system (see Section 2.2.9 on page 14) where the server core is responsible for managing all the pools. If a module wants to allocate memory, it needs to ask the core for permission by using the API.

The API also contains functions for array manipulation, table manipulation, string manipulation, identification, authentication etc.

2.2.8 Configuration Parameters Affecting Performance

The Apache server is known to handle heavy load situations quite well, but it can not optimize its performance by itself. That is up to the administrator to do by tuning certain directives [16] in the server. Performance tuning is usually the art of trading off one resource against another, to get a

good balance between CPU time vs. memory space or fast response vs. large throughput. The single most important choice an administrator needs to make is what MPM (see page 7) to use. Depending on the operating system this can affect the server's performance a great deal. Of all the changeable parameters some are directly related to performance, others are not that obvious. These directives can be divided into three different categories which specifies what area of the server the parameter affect. The three categories are: process management, network management and HTTP management. The changeable parameters for the Prefork MPM have already been mentioned in Section 2.2.4 and the ones for persistent connections in Section 2.2.4. The parameters of the former belongs to the process management category and the latter to the HTTP management category. All of those directives affect the performance of the server.

Process Management

MaxClients An increase of the *MaxClients* parameter allows the parent process to create more child processes, which will make the server able to handle more clients at the same time. This should therefore increase the server's throughput. Each new child process handling a client's request will require memory space and CPU time, so the average time Apache spends on a client's request (the *Apache service time*) might increase. This will not necessary mean that the average response time for the clients will increase, because the end-to-end response time is highly dependent on the amount of clients in the backlog queue (see Section 2.3.6 on page 17). If for example the server is idle and suddenly 15 clients try to connect to it at the same time, with the *MaxClients* parameter set to 5, 5 of them will start to be processed by the server while the other 10 will be put in the backlog queue. Depending on how long time these requests stay in the queue it might be better for the end-to-end response time to use a *MaxClients* value of 15 and process all the client's requests at the same time. Even though this might lead to an increased average *Apache service time* as a result of the increased use of system resources.

A lower value of *MaxClients* decreases the number of child processes allowed in the system. This will free up memory space and CPU time, which in turn should improve the *Apache service time*. but reduce the throughput.

Care has to be taken not to let the server consume all of the available memory. If this happens the computer will need to swap some of the server's memory pages onto the hard-drive [18]. This will increase the CPU load and degrade the server's performance in all variables. An idle child process still consumes memory space, so if the *MaxClients* value is set too high it may cause the server to start swapping even under light server load.

The general rule is to set the value of *MaxClients* as high as possible without causing the computer to start swapping memory pages onto the disk.

ServerLimit This parameter works as the upper limit for *MaxClients*. A change in this parameter may in turn change the *MaxClients* parameter with all its implications described above. The reason why the *ServerLimit* directive limits the *MaxClients* value is because it specifies how many slots that should be initialized when creating the scoreboard. This parameter therefore decides how much shared memory space that will be used by Apache. If this parameter is assigned a too high value, a lot of unused memory will be allocated, which instead might have been better used by the server's child processes while serving requests.

StartServers The *StartServers* value only affects the performance at server startup. If the value is much less than the number of clients trying to connect to the server, it will take a while before the parent process will have managed to create all the child processes needed to serve all the clients. This will lead to long response times and a small throughput at server startup. On the other hand, if the *StartServers* value is too high, a lot of unnecessary child processes will be created, which means a higher memory usage.

MinSpareServers If this value is too high an unnecessary amount of idle child process will exist, which will consume memory space. On the other hand the server will be able to react quickly

on a large amount of simultaneously incoming requests, if there suddenly would be a peak in the server load. If set too low, there will not be enough idle servers ready to serve incoming requests, which means that the parent process needs to create these child processes before the requests can be taken care of. This will increase the CPU load, and the response time.

MaxSpareServers This value should be set higher than *MinSpareServers* to have any effect. Setting this parameter too high will allow a large number of idle child processes to be created which leads to the same implications as mentioned above. This value is used to lower the amount of idle child processes after a peak in the server load.

MaxRequestsPerChild As mentioned previously this parameter's main task is to guard against memory leaks. When this parameter is assigned a lower value the lifetime of each child process is decreased. Therefore if set too low it will mean that running child processes will be terminated often and new ones have to be created. This will increase the CPU load. Like *MaxSpareServers* this parameter might help to thin out the number of running child processes after the server has been through a busy period.

MaxMemFree This parameter limits the amount of memory which Apache is allowed to hold ready for use. The amount is specified in kilobytes (kB). After Apache has retained an amount of memory from the operating system to be able to serve a request, it does not return the memory. Instead Apache keeps it in an internal memory pool, which will continue growing in order to satisfy future memory demands. To retrieve the required memory from the internal pool is faster than retrieving it from the operating system. The fact that Apache does not return the memory to the operating system might lead to problems when the server has been through a busy period. During this period Apache might have retrieved a lot of memory to the internal pool which is not needed anymore. This memory will therefore be unused and unavailable to other applications on the computer.

It is beneficial for the performance of the server to have some memory ready, but if the amount is too high it might degrade the performance of the underlying operating system.

Network Management

ListenBackLog This parameter limits the size of the backlog queue, i.e. the number of requests that are queued up when all of the servers child processes are busy (see Figure 2.4 on page 18). It is necessary to have a queue to be able to handle a heavy server load. If the queue is allowed to be very large the server will be able to handle a larger amount of requests during a busy period. This should increase the throughput. The problem when the queue starts growing is that the response time for a client's request might become very long, because the average time each request spends in the queue will increase.

If this queue is limited and made very small, the server will not be able to handle a load peak without blocking some of the client requests.

SendBufferSize With this directive it is possible to change the size of the *connection socket's* output buffer. This is mainly useful when the round-trip time for a connection is long, because it makes it possible for Apache to queue more data.

The value of this parameter affects all the socket buffers created by Apache, so if it is set too high the memory usage might become critical during busy periods.

HTTP Management

KeepAlive If *KeepAlive* is used the dialog between a client and the server will run faster, because the server will not close down its connection to the client between requests. If clients sends a lot of requests to the server, using *KeepAlive* will decrease the CPU load a lot. The problem with using *KeepAlive* is that the child processes can not serve any other clients until the

current client disconnects, which in some cases will limit the throughput. The following two directive regulates this.

KeepAliveTimeout If a larger time slot is used more requests from the same client will be able to use the same connection. It also means that it will take longer time before a child process is available to serve another client's request. If the value of *KeepAliveTimeout* is too high during a busy period, all the child processes will be serving clients over persistent connections. It will take quite a long time before any of these connections times out, which means that the number of requests in the backlog queue will start growing.

MaxKeepAliveRequests This directive can be used to force a client to release its connection with a child process, if it has been active for too long. For example if a client issues a large number of request with short intervals it may stay in the *KeepAlive* loop forever.

TimeOut This parameter actually specifies three different types of timeout values in connection with the established HTTP connection:

- the time from connection being established until receiving GET. This timeout does not affect the persistent connections because they use the *KeepAliveTimeout* value for this instead.
- the time since last packet of data was received on a PUSH or PUT HTTP request
- the time since last acknowledgment (ACK), if the server is waiting for more data from the client

Here is a list of other configuration directives that might have an impact on the performance: ScoreBoardFile, RLimitNPROC, RLimitMEM, RLimitCPU, ReceiveBufferSize, LogLevel, LimitRequestLine, LimitRequestFieldSize, LimitRequestFields, LimitRequestBody, mod_deflate. See [6] for more information.

2.2.9 Memory Pools

To avoid the problems with memory leakage Apache provides resource pools. Every resource like memory, open files etc. will be connected to a pool. The resources will automatically be released when the server is finished with the pool. For example, each request has its own pool to keep its resources in. As soon as the server is finished serving the request all the resources connected to the pool will be released.

2.2.10 Why Use Prefork?

Processes unlike threads, have separate address spaces and do not share memory or resources directly. Therefore processes consume more system resources than threads and will load the server more. To switch between processes, which is called a context switch, is usually a heavier task than to switch between threads. The problem with prefork is that a child process is needed for each request, which consumes a lot of system resources. There are other MPM:s for Linux e.g. *worker* and *perchild*, where each child process keeps many threads to serve requests. The problem with these multi-process multi-threaded servers is that many of the third-party modules you need to include for a website (e.g. PHP) are not thread-safe under Linux/Unix. Stability and performance are the most important things concerning a website and at the moment none of the other MPMs can compete with Prefork in terms of that. If an error occur with a request when using the Prefork MPM the problem is encapsulated to one process and will not affect the other requests, which may not be the case when using a multi-process multi-threaded server.

2.3 Internet Protocol Suite (TCP/IP)

The Internet Protocol suite is the most common way to establish a network connection between two clients today. The Transmission-Control-Protocol (TCP) and the Internet-Protocol (IP) are its two most popular protocols which explains why this suite is often called the TCP/IP suite. A more complete description can be found in [19][20][21].

The Internet Protocol Suite is made up of five layers: physical, data link, network, transport and application. In Figure 2.2 there are some examples of what protocols you can find in the different layers.

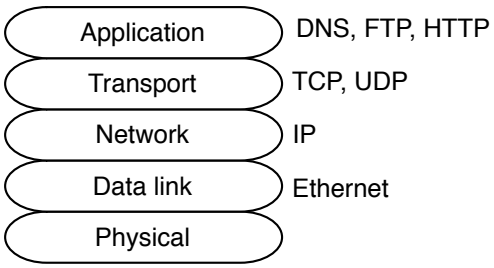

Figure 2.2: The five layers in the Internet Protocol Suite and examples of some of the protocols you might find in the different layers.

Physical layer is responsible for transmitting data as a bit stream over a physical medium [19].

Data link layer's main task is to move data between the network interfaces of two different computers on the same physical network. A protocol defined at this layer is called a node-to-node protocol where, in the case of using Ethernet, each node is identified by a unique MAC-address [20].

Network layer makes sure that the data is delivered to the right host when the host is not on the same physical network. The Internet-Protocol (IP) is used for doing this and is therefore called a host-to-host protocol where each host is uniquely identified by its IP-address.

Transport layer is responsible for delivering the data to the right process on the host. It uses a unique port address (see Section 2.3.3 about TCP Ports below) on the host to identify the process. A protocol defined at this layer is therefore called a port-to-port protocol.

Application layer contains the applications/processes that makes it possible for a human user to get access to the network.

2.3.1 Apache and TCP/IP

The client's browser and the Apache server uses TCP, to establish a connection and transfer data between each other. The data is sent in small chunks called packets. TCP uses IP to send single packets to the correct destination. IP is a best-effort protocol, which means that it tries to send the packets the best it can but without any guarantee. Packets sent may or may not be received at the final destination and will most likely not arrive in the same order as they were sent. When reliability is important, the IP protocol must be paired with a reliable network protocol like TCP. TCP assigns each packet with a specific sequence number to be able to guarantee in-order delivery. When a packet is received the receiver will send an acknowledgment, ACK, back to the sender, containing the next sequence number it expects to receive. The sender will then be sure that the

packet has been delivered and at what sequence number the receiver is at. To make it possible for the sender to not have to wait for an acknowledgment for each single packet before transmitting the next packet, TCP defines an amount of bytes (called a transmission window [20]) that it may send before expecting an ACK from the receiver.

2.3.2 Round-Trip Time

The TCP sender estimates the time it will take to send a TCP segment and receive a reply to that segment, this estimate is called round-trip-time (RTT). The estimate is based on measurements of previous sent segments. The round-trip-time is part of TCP's congestion control [22] [20] and is used to calculate the retransmission timeout (RTO). If the sender has not receive an acknowledgment for the data it has sent before the RTO it will regard the data as lost and will retransmit the data. Here is how the RTO value is calculated [22]:

$$\begin{aligned} RTTVAR &= \frac{3}{4} \cdot RTTVAR + \frac{1}{4} \cdot \mid RTT - R \mid \\ RTT &= \frac{7}{8} \cdot RTT + \frac{1}{8} \cdot R \\ RTO &= max(RTT + 4 \cdot RTTVAR, 1s) \end{aligned}$$

where R is the last measured round-trip time, RTT is the smoothed mean round-trip time based on the recent measurements and RTTVAR is the mean variance of the RTT value.

2.3.3 TCP Ports

TCP uses ports to identify different applications, e.g. a HTTP server usually runs on port 80. To send a request to the server you specify the server's IP-address and the port number of the server. Every client application sending the request will also have a unique port number assigned to its request. This makes it possible for the client to establish multiple connections to the same server.

2.3.4 Establish A Connection

The procedure which this protocol uses to establish a connection is called the 3-way-handshake.

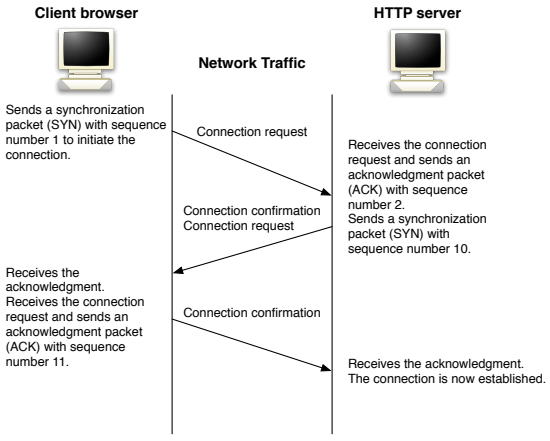

Figure 2.3: An example of how TCP establishes a connection by doing a 3-way-handshake.

Here's an example of how it works (see Figure 2.3).

1. The client sends a connection request, in form of a SYN packet, to the server. The SYN packet is used to synchronize sequence numbers between the client and the server. E.g. this first SYN might contain the number 1.

2. The server receives the request and sends a SYN and ACK response to the client. The ACK will contain the next sequence number that the server expects to receive, in our example number 2. The SYN packet will contain the initial sequence number for the server, e.g. 10.

3. The client then sends an ACK response to the server. The ACK will contain the next sequence number that the client expects to receive from the server, in our example 11.

The connection has now been established, because both the server and the client have received an acknowledgment of it.

2.3.5 TCP Sockets

The port number and IP address of the host is used to create a unique identifier for each network application. This identifier is called a *socket* and is used by the application to send and receive data over the network. A detailed description of how this is done in Linux can be found in [23]. The socket is also connected to a specific transport protocol, which in the Apache server's case is TCP. The Apache HTTP server use two different types of TCP sockets:

Listen socket which is used to establish a connection with the client. It might also be referred to as *dialup socket* or *server socket*.

Connection socket The server uses this socket type for data transfer with the client.

Figure 2.4 shows an example of how the Apache HTTP Server uses these two kinds of sockets to communicate with the client's browser. On the server side in this example there is just one *listen socket*, which is connected to the server's TCP port (usually port 80 for HTTP servers). Only one child process (called the *listener*) is allowed to use the *listen socket* at a time. The client's browser uses a *connection socket* to establish the connection with the *listen socket*. When the 3-way-handshake is completed the client's request is placed in the *listen socket's* backlog queue (sometimes also referred to as the accept queue because the system call used to remove the request from the queue is called *accept*). The child process connected to the *listen socket* will be informed about the request and remove it (accept it) from the backlog queue by using the system call *accept* (see the Linux manual page: *man 2 accept*). The *accept* system call will create and return a *connection socket* connected to the client, which the child process may use to communicate with the client. The child will then release the *listen socket* so that one of the other idle child processes may use it to listen for the next incoming request. When this has been done the child process starts processing the client's request through the *connection socket*. When the child process has finished serving the request (and when the *KeepAlive* functionality has timed out if used) this process will queue in for the *listen socket* among with the other idle child processes.

When using the Apache HTTP server there are no restrictions to using only one single TCP port. In the configuration file *httpd.conf* you can actually specify multiple ports for which you want the server to be connected to. If you want the server to listen for incoming connections on both port 80 and port 8000 you use the configuration directive *Listen* like this:

```
Listen 80
Listen 8000
```

For each one of those ports a *listen socket* will be created and therefore multiple child processes may act as a *listener* at the same time. When an idle child process becomes a *listener* the Prefork MPM uses a round-robin scheme to decide what listen socket it should use.

2.3.6 The Backlog Queue

There might be times when all the child processes are busy serving requests and in that case there will not be any *listener* process. When this happens the incoming requests to the server will still end up in the backlog queue.

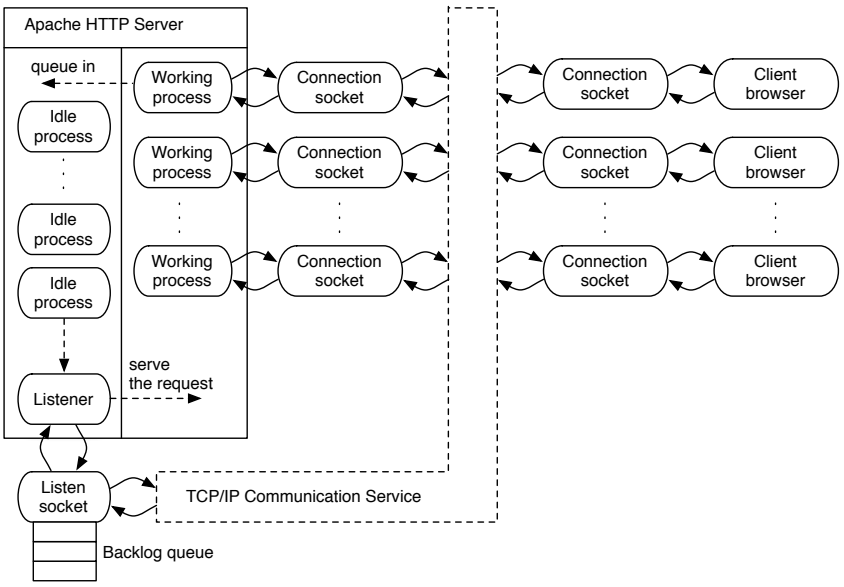

Figure 2.4: A simplified model of how the Apache HTTP Server and client's browser use sockets to communicate with each other.

Example: Consider a system using the Apache server with *MaxClients* set to 50 and persistent connections (*KeepAlive*). If all of the 50 child processes are busy when a request is sent to the server, the connection is still established but can not be accepted at once by Apache. It is therefore put into the backlog queue of the *listen socket*. As soon as any of the 50 persistent connections times out, it will accept the request and handle it. The backlog queue is designed so that the first request which enters the queue, will be the first one served, a so called FIFO-queue (First-In-First-Out).

The backlog queue makes it possible for the server to handle a sudden increase in traffic, a so called load spike (or sometimes also referred to as "flash crowds"). During the heavy load the backlog queue will fill up because the working child processes will not be able to serve the requests in the same rate as they enter the system. Later when the load decreases the server might be able to empty the queue and therefore still be able to serve all the requests.

The size of the backlog queue is therefore very important for the performance of the server. On the Linux platform I was running my server the size of this queue had a default maximum value of 128. To see what your computer's max value is you can type the following command in a terminal window (see Section 2.3.7):

```
$ cat /proc/sys/net/core/somaxconn
128
```

2.3.7 Netstat

To get information about the different TCP connections to a server it is possible to use the command line tool *netstat*. If *netstat* is typed in a terminal window it will show information of every single network connection the computer has. If *netstat -t* is typed in the terminal window only the TCP socket connections are shown. When a *p* is added to the options string like this *netstat -pt*, it will also display the process id (PID) and name of the program which each socket belongs to. Here is an example of how *netstat* can be used:

Example: For this example the Apache server was running on port 8000 with *MaxClients* set to 1, so it just could handle one request at a time. The host name of the computer the server

was running on was: *ragazzini.control.lth.se*. A browser on the same computer was used to send requests to the server. Two requests were sent at the same time to the server, and this was the result from *netstat*:

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	ragazzini.control.lth.:8000	ragazzini.control.lth:40926	ESTABLISHED	14625/httpd
tcp	441	0	ragazzini.control.lth.:8000	ragazzini.control.lth:40929	ESTABLISHED	-

From this data we can see that two requests have been sent to the Apache server from the same host. The connection socket for the first request is working at port 40926 and for the second one at port 40929. Both requests have the status *ESTABLISHED* but it is only the request from port 40926 that is being served by the Apache's httpd process. This is discernible by looking at the PID/Program name column. The process id for the child process handling the request is 14625. The other request have not been assigned a program, so it has been placed in the backlog queue waiting for the child process 14625 to accept it. It is also possible to figure out what requests are already being served or not by looking at the Recv-Q column. It specifies how many bytes that have not yet been copied by the program connected to the socket. For the request at port 40929 in the previous print out there is still some data waiting to be copied which means that this request has not been taken care of.

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	ragazzini.control.lth.:8000	ragazzini.control.lth:40929	ESTABLISHED	14625/httpd
tcp	438	0	ragazzini.control.lth.:8000	ragazzini.control.lth:40939	ESTABLISHED	-

When this print out from *netstat* is made, the server has finished handling the request from port 40926 and is now working with the other request from port 40929. Because just one child process was used, the PID of the process serving the request is still 14625. You can also see that I've made another request from the browser that has gotten the port number 40939 which is now waiting in the backlog queue to be served. There is no information displayed about the *listen socket*. The Apache server was configured to listen on port 8000, so there should be at least one listen socket. The reason why the listen sockets do not appear is because *netstat* does not display them as default. To show the listen sockets you have to use either the *-l* flag which makes *netstat* just show the *listen sockets* or the *-a* flag, which will show all the sockets. Here is a print out of the listen socket information:

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	*:8000	*:*	LISTEN	14624/httpd

The *:8000 means that the server is listening on port 8000 for all IP addresses associated with the computer the server is running on. If you want the server to be more specific you can specify an IP address along with the port number with the *Listen* directive:

```
Listen 130.235.83.17:8000
```

Tuning the Settings For TCP sockets

The settings you can change for the TCP sockets in Linux are available under the folder */proc/sys/net/*. For instance the max length of the backlog queue is specified in the file */proc/sys/net/core/somaxconn*. Under */proc/sys/net/ipv4/* you will find more settings relevant to the TCP sockets. To change the values in these files you need to have root privileges on the computer. If you do not have that or just want to change the settings for one particular socket you can use the system call *setsockopt*. To see what options that are available to set with this system call have a look at the Linux manual page: *man 7 tcp*. There also exists a system call called *getsockopt* which you can use to see how the available TCP options are set for a particular socket.

2.4 The Linux Process File System

The Linux process file system [24], [18], also known as the proc file system and sometimes called the process information pseudo-file system, is a file system where a lot of useful information about processes running on the computer can be found. Each running process has its own directory under the root directory /proc, and its process id, PID, is used as the name of the directory. In this directory you will find files containing various information about the process. This information is not stored anywhere but is computed on demand and presented as plain text when a user tries to read them. Apart from the process information, the proc system also contains files with global statistics about the computer, e.g. memory usage, performance statistics, kernel version and data about loaded drivers. A lot of system calls in Linux, e.g. *ps*, parses relevant proc files to display its data. One Section of the proc system i.e. /proc/sys is dedicated to kernel variables. An administrator can read and write to these files. To tune a kernel parameter the administrator writes the new values to the appropriate file. The proc system is the control and information centre of the kernel.

2.5 Performance Issues with Web Servers

The two most important metrics when talking about web server performance are response time and throughput. The response time is the time it takes from that a client sends a request to the server until it receives a response, the end-to-end time. This time can be divided into network time (transporting the data between the hosts) and web server time (the time it takes for the server to handle the request). In this thesis I am going to neglect the performance issues related to the network and therefore not regard the network as a possible bottleneck. The response time is probably something that would be good for a controller to have access to and make decisions depending on its value. In real life it would be impossible to measure the end-to-end response time because the controller would not have access to the client's side. If the network issues are neglected the average time Apache spends on a request (the *Apache service time* will be a good indication of what is happening with the end-to-end response time.

The throughput is usually measured in the number of requests per second which the server is able to handle. My main focus will be on the server, so I am interested in what may cause the performance to degrade on the server side.

The server consists of the server software (The Apache HTTP Server), the operating system (in my case Linux), hardware platform (processes, memory, harddrives, etc.) and the contents which the server provides. All of these different parts play important roles when it comes to the server's performance.

The most common performance issues with web servers are related to insufficient bandwidth at peak times, overloaded servers, uneven server loads, delivery of dynamic content, shortage of connections between application servers and database servers, failure of third-party services and delivery of multimedia contents [25]. In this thesis I am going to neglect the performance issues related to the network and instead just focus on what can be done on the server side to improve the performance.

2.6 Workload generation

When running tests on the server system to do e.g. capacity planning or changing a directive to see its impact on the performance, there needs to be some traffic load on the system. The system will react quite differently depending on the load so it is important to generate a representative workload.

The problem with web server traffic is that it is known for being highly variable and even have self-similiar properties[2] [26] which is known to have a negative impact on the performance. When the traffic to the server is varied it will result in high variability on the server in CPU load, number of open connections, memory usage, hard-drive usage etc. To capture all these features in a generated synthetic load it is possible to use either a trace based workload or an analytic workload. Trace based workload is real recorded traffic to a server that can be replayed and be used to load a server system. The good thing with this is that this traffic is a realistic traffic. The problem is that we can not be certain that all features of the traffic have been captured during the recording session. Because web traffic is self-similiar it might take a very long time, to be able to see that feature in the recordings. This problem is not an issue when using analytic workload. An analytic workload is based on mathematical models that describe different workloads characteristics. The problems with this method is to identify those characteristics that are important to model to get a reasonable web traffic load. Information about generating representative workload can be found in [27] and [25].

[2]Self-similarity means that a small part of an object has the same structure as the whole object. This is a property present in e.g. fractals. For network traffic this implies statistics with long-tails distribution.

Chapter 3

Actuation for Control

3.1 Introduction

In order to control the Apache web server with consideration of the performance metrics mentioned in Section 2.5 it is a must to be able to influence the server in some way. Because Apache is a software program this means to be able to update important configuration directives while the server is running and make it pick up the changes. Once this is possible these directives could be used to create a controller for the server.

As shown in Section 2.2.8 (on page 11) there exist quite a lot of performance related directives, so the question is which ones of these should be chosen? To use all of them would be unnecessary and make the controller far too complex. Some of these parameters affect the server more than others, so focusing on the parameters that really makes a difference is the best option.

In Section 2.2.8 the different performance directives were divided into three different categories: Process management, Network management and HTTP management. All of these three groups are important for the server's performance so if the most relevant parameters from each group are chosen it is possible to have quite good tools to change the behavior of our server during runtime.

Process management All directives in this group regulate the number of child processes in some way. The most important ones of these are *MaxClients*, *MinSpareServers* and *MaxSpareServers*. There is no parameter that can be used to tell Apache exactly how many child servers it should be running,

Network management The *ListenBacklog* is by far the most important directive in this group, because it sets the size of the backlog queue for the *listen socket* (see Section 2.3.6 on page 17).

HTTP management It is hard to find a reason for ever wanting to set the *KeepAlive* parameter to *Off*, which means turning off persistent connections. Very seldom anything is gained by doing so and therefore it will always be kept on without the possibility to turn it off. The most important parameter in this group is *KeepAliveTimeout* and to be able to change its value during runtime would be beneficial.

In the introduction there was a discussion about a simplified model of a web server and three parameters were mentioned that would be beneficial to be able to update: Number of servers (i.e. child processes), queue length and queue timeout. The "Number of servers" are connected to the process management group, and the "queue length" correspond to the *ListenBackLog* directive. The only parameter not represented in Apache is the "queue timeout" parameter.

Now when the directives have been chosen, it is desirable to be able to update them during runtime.

3.2 Influence the Process Management

The two directives *MinSpareServers* and *MaxSpareServers* are used by Prefork's own heuristic control to regulate the number of idle child processes in the server (see Section 2.2.4). To make sure that this control does not lead to an infinite number of child processes, the *MaxClients* value is used as a limit for the total number of processes. If these three values were used for a controller, it would rely on the Apache's underlying heuristic control. Because there are three parameters to update it might be quite complex to figure out how to change these values to achieve better performance. Therefore it is desirable to use a more easy and direct control of the number of child processes as the one mentioned in the introduction. Instead of just regulating the number of idle processes a controller should be able to regulate the total number of processes by updating just one parameter. This would add more power and flexibility to the controller. To be able to do this, Prefork's own spare servers control has to be disabled. One way of doing this would be to set both *MinSpareServers* and *MaxSpareServers* to zero. The number of child servers would then not be regulated at all, but just continue to grow until it eventually would hit the *MaxClients* value (see Figure 3.1).

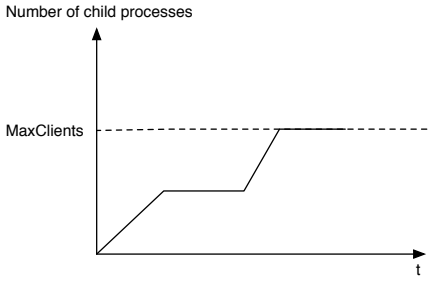

Figure 3.1: The situation which will occur when Apache's spare server control has been disabled.

The *MaxClients* value could then be used to regulate the total number of processes in the system. The problem with this approach is that it will lead to that the benefit of having idle child processes ready and waiting for incoming requests will be lost. Because an increase in the *MaxClients* value will not necessary mean that new child processes are created at once, just that more child process can be created. As long as all the existing child servers are busy and as long as the number of child servers has not reached the *MaxClients* value it will end up in a state where each new client's request will lead to a new child process being created no matter how high the *MaxClients* has been set. This way of controlling the server will be problematic and not very flexible. What is gained with this approach is that it would still be possible to use Apache's own *MaxClients* limit control, but now when the idle process control has been disabled this is not much of a benefit. If the spare servers control in Apache is not going to be used, it is probably best to discard this parameter as well.

As mentioned previously it is desirable to make the system as flexible as possible for the controller. One way of doing this is by letting the controller use one single parameter to specify the total number of child servers that should be in the system. Because no such parameter exists it has to be created.

3.2.1 The New Parameters

For this to work two new parameters will actually be needed: one that will keep track of the number of running child processes and another that will contain the number of running child processes that should be running. The former will be called *running*, and the latter will be called *want_running*. It is also desirable to have a third parameter which should keep track of the number of idle child processes. This parameter will be called *idle_running*. Even though three parameters are now being used it will not lead to the same complexity as before. Now the controller will just

be using one parameter to change the total number of child processes, i.e. by updating the value of *want_running*. The other two parameters will be updated automatically by the system and just be used by the controller as read-only information parameters, which it will use to base its decisions on. Now when the new parameters have been specified, it is time to add them to the server.

To make it possible for a controller to change the number of processes during runtime, it is desirable to be able to update these parameters in realtime. At the moment it is sort of possible because you can update a parameter in the configuration file httpd.conf and then do a graceful restart of the server, and the child processes will eventually update their configurations. The problem with this is that it might be quite inefficient, if a lot of changes need to be done during a short period of time. Especially when the server is busy, because then a lot of child processes need to be killed and created. As new parameters are going to be added to the source code and a new process management control will be created, the code to update these new parameters might as well be added directly to the source code.

3.2.2 Implementation of the New Directives *running*, *want_running* and *idle_running*

These new parameters are all global parameters which both the parent- and any child process should to be able to read and update. Therefore it makes sense to put these values in the shared memory space of the scoreboard (see Section 2.2.4 on page 8). For the *running* parameter this is done by adding the following data structure called *running_info* to the *include/scoreboard.h* file:

```
typedef struct running_info running_info;
struct running_info {
    int                 running;     // nbr of child-processess running
    apr_proc_mutex_t    *mutex;      // must be owned to access the above field
};
```

When a process starts or dies it calls a function in the scoreboard in the file *server/scoreboard.c* called *ap_update_child_status_from_indexes* to change its status. All the different statuses that a child process might have are defined in the the *include/scoreboard.h* file like this:

```
#define SERVER_DEAD 0
#define SERVER_STARTING 1        /* Server Starting up */
#define SERVER_READY 2           /* Waiting for connection (or accept() lock) */
#define SERVER_BUSY_READ 3       /* Reading a client request */
#define SERVER_BUSY_WRITE 4      /* Processing a client request */
#define SERVER_BUSY_KEEPALIVE 5  /* Waiting for more requests via keepalive */
#define SERVER_BUSY_LOG 6        /* Logging the request */
#define SERVER_BUSY_DNS 7        /* Looking up a hostname */
#define SERVER_CLOSING 8         /* Closing the connection */
#define SERVER_GRACEFUL 9        /* server is gracefully finishing request */
#define SERVER_IDLE_KILL 10      /* Server is cleaning up idle children. */
#define SERVER_NUM_STATUS 11     /* number of status settings */
```

The *running* parameter

When a child process starts it will have the status *SERVER_STARTING* and when it dies it will have the status *SERVER_DEAD*. It is when this happens that the *running* parameter should be increased or decreased. When more than one process want to update this parameter at the same time a synchronization problem like the one in Figure 3.2 might occur.
In this example one process is starting and wants to increase the *running* parameter. It manages to read the value of the parameter which is 6 and stores this value locally, but do not have time to update it before a context-switch occur and the process is switched out. The other process that is switched in is ending. It reads the *running* parameter which still has the value 6 and manages to update the *running* value to 5 before it is switched out. When the starting process then continues

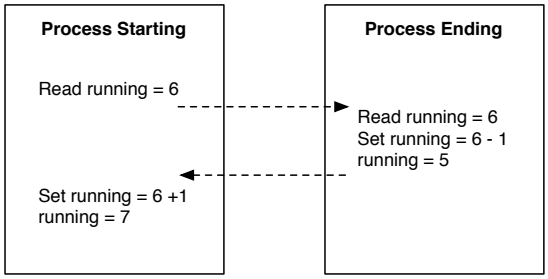

Figure 3.2: The parameter running should be 6 but end up as 7.

from where it was switched out, it had just read the value of *running* and stored it. It does not know that the other process has updated the value to 5. Instead it assumes that the value of *running* still is 6, and increase the value to 7, when it really should increase the value to 6.

To guarantee that this scenario will not happen mutual exclusion is needed [28], and that is the reason for having the *mutex* parameter in the *running_info* struct. Before a process is able to read from or write to the *running* parameter it must first get the ownership of the *control_info* struct by locking the *mutex* to the process. When it manages to do that, no other process can read or write to the parameter until the owning process releases the *mutex*. How this is done is also available in the flowchart in Figure 3.7 on page 30.

The *running_idle* parameter

The same synchronization problem might occur with the *running_idle* parameter, because it is possible that many child processes want to update its value at the same time. So to avoid this problem a similar data structure is added for this parameter to the *include/scoreboard.h* file:

```
typedef struct running_idle_info running_idle_info;
struct running_idle_info {
    int                 running_idle;// nbr of idle child processess
    apr_proc_mutex_t    *mutex;      // must be owned to access the above field
};
```

The *running_idle* parameter will also be updated from the *ap_update_child_status_from_indexes* function in the file *server/scoreboard.c*. A child process is said to be idle when it has the status *SERVER_STARTING* or *SERVER_READY*. As long as a child process lives it will just get the status *SERVER_STARTING* once, and when this happens the value of *running_idle* should be increased. After the server startup has finished the child's status will change to *SERVER_READY*, which also is an idle status. This however should not lead to an increase in the *running_idle* value because this will already have been done when the child process was starting as mentioned above. A child might get the *SERVER_READY* status more than one time during its lifetime. When this happens and the old status was not *SERVER_STARTING* the value of *running_idle* should be increased. For instance it will go back to this status when it has finished serving a request and the *KeepAliveTimeout* has been reached (if *KeepAlive* is being used). In the *ap_update_child_status_from_indexes* function the child's old status is available, so it is easy to see what status a child leaves. Every status is also assigned an integer value (these values are available at the beginning of this Section) which can be used to examine how to update the *running_idle* parameter. The value of *running_idle* should just be increased if the child process changing status to *SERVER_READY* is leaving an old status that has a higher integer value than *SERVER_READY*. With the same reasoning the *running_idle* value should be decreased if the child leaving the status *SERVER_READY* is about to get a new status that has a higher integer value than *SERVER_READY*. How the update of the *running_idle* parameter should be done is also available in the flowchart in Figure 3.7 on page 30.

The *want_running* parameter

Synchronization is not an issue with the *want_running* parameter, because it will just be updated from one place i.e. the controller. It does not need to be wrapped in another data structure, instead a standard integer variable is used.

Adding The Parameters To The Scoreboard

To make these new parameters available to the parent and all of the child processes they are added to the scoreboard's global data structure (see Section 2.2.4) called *global_score* in the *include/scoreboard.h* file:

```
typedef struct {
    int              server_limit;
    int              thread_limit;
    ap_scoreboard_e  sb_type;
    ap_generation_t  running_generation; /* the generation of children which
                                          * should still be serving requests. */
    apr_time_t  restart_time;
    running_info  running_info;
    running_idle_info  running_idle_info;
    int  want_running
} global_score;
```

In the *server/scoreboard.c* file the following code is added to the *ap_create_scoreboard* function to assign default values to the new parameters *running* and *mutex* in the *running_info* struc:

```
ap_scoreboard_image->global->running_info.running = 0;
apr_proc_mutex_create(&(ap_scoreboard_image->global->running_info).mutex,
            "running_info", APR_LOCK_DEFAULT, p);
```

The variable in Apache containing the scoreboard is called *ap_scoreboard_image*. It contains pointers to the global, parent and servers data structures (see Section 2.2.4). The function *apr_proc_mutex_create* is one of the functions in the Apache Portable Runtime library (see Section 2.2.6) and is used to create a *mutex* variable. When the *mutex* has been created for the *running_info* struct, this is what a process needs to do in order to update the *running* parameter:

```
apr_proc_mutex_lock(ap_scoreboard_image->global->running_info.mutex);
ap_scoreboard_image->global->running_info.running--;
apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
```

The function *apr_proc_mutex_lock* tries to lock the *mutex* to the current process and get ownership over the *running_info* data structure. As soon as it gets this, the process may update the *running* parameter. The function *apr_proc_mutex_unlock* then releases the process' lock on the *mutex*, and therefore its ownership of the data, so other processes may get access to the *running_info* data structure.

To assign a default value to the *running_idle* parameter the same thing is done for the *running* parameter. In the *server/scoreboard.c* file the following code is added:

```
ap_scoreboard_image->global->running_idle_info.running_idle = 0;
apr_proc_mutex_create(&(ap_scoreboard_image->global->running_idle_info).mutex,
            "running_idle_info", APR_LOCK_DEFAULT, p);
```

Now in order to update the *running_idle* parameter a process needs to do the same thing as for the *running* parameter.

The *want_running* parameter is not assigned a default value at the same place as the other two parameters above, because the *MaxClients* value from the *httpd.conf* file is chosen to be used as the default value. The reason for this is to get the possibility to set a default value for this parameter in the configuration file and to use an existing configuration parameter for this purpose saves time. In the source code, the parameter that holds the *MaxClients* value is called *ap_daemons_limit*.

Therefore the following code is added to Prefork's run function *ap_mpm_run* which is located in the file *server/mpm/prefork/prefork.c*:

```
ap_scoreboard_image->global->want_running = ap_daemons_limit;
```

3.2.3 The New Server Maintenance Function

Now with the use of the two new parameters *running* and *want_running* it will be easy to implement a new server maintenance function. The purpose of Prefork's idle server maintenance was to keep the number of idle child processes within a certain limit. The purpose with the new version of the server maintenance will be to keep the number of child processes equal to the *want_running* parameter. The idle server maintenance was performed in the function *perform_idle_server_maintenance* in the file *server/mpm/prefork/prefork.c*. This function will be overridden with the new server maintenance code. Here is a description of how this function is now supposed to work:

- If the value of the parameter *running* is smaller than the value of *want_running* the parent process should create as many child processes as necessary to make these two values equal.
- If the value of the parameter *running* is larger than the value of *want_running* the parent process should kill enough of child processes to make these values equal.

To create a new child process the parent needs to specify a slot in the scoreboard that is free to use. Instead of doing like Prefork, which loops through the scoreboard slots until it has found enough free ones, it is possible to improve performance by letting the scoreboard keep track of all the free slots itself. To make this possible two more parameters are added to the scoreboard. The first parameter will be called *first_free* and be a global parameter that will contain the index to the first free slot in the scoreboard. The other parameter called *next_free* will be added to each slot in the scoreboard, and will contain the index to the next free scoreboard slot that comes after the slot you are on. After the initialization of the scoreboard the slot setup with these two new variables will look like the one in Figure 3.3. This is a so called single-linked list which is linked together by the slots' *next_free* parameter.

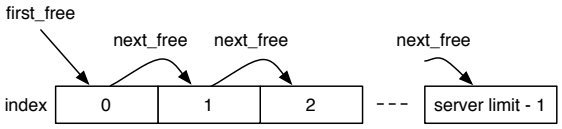

Figure 3.3: The default setup for the scoreboard slots with the new parameters *first_free* and *next_free*.

The *first_free* directive is added to the running_info struct, because it will be used in connection with the *running* parameter defined there which is shown below. This is the updated running_info struct:

```
typedef struct running_info running_info;
struct running_info {
    int                 running;    // nbr of child-processess running
    int                 first_free; // index of the first free scoreboard slot
    apr_proc_mutex_t    *mutex;     // must be owned to access the above fields
};
```

With the use of the variables *first_free* and *next_free* the flowchart for the new server maintenance will look like the one in Figure 3.4. The implementation of this function is available in *C* code in Appendix B.1.

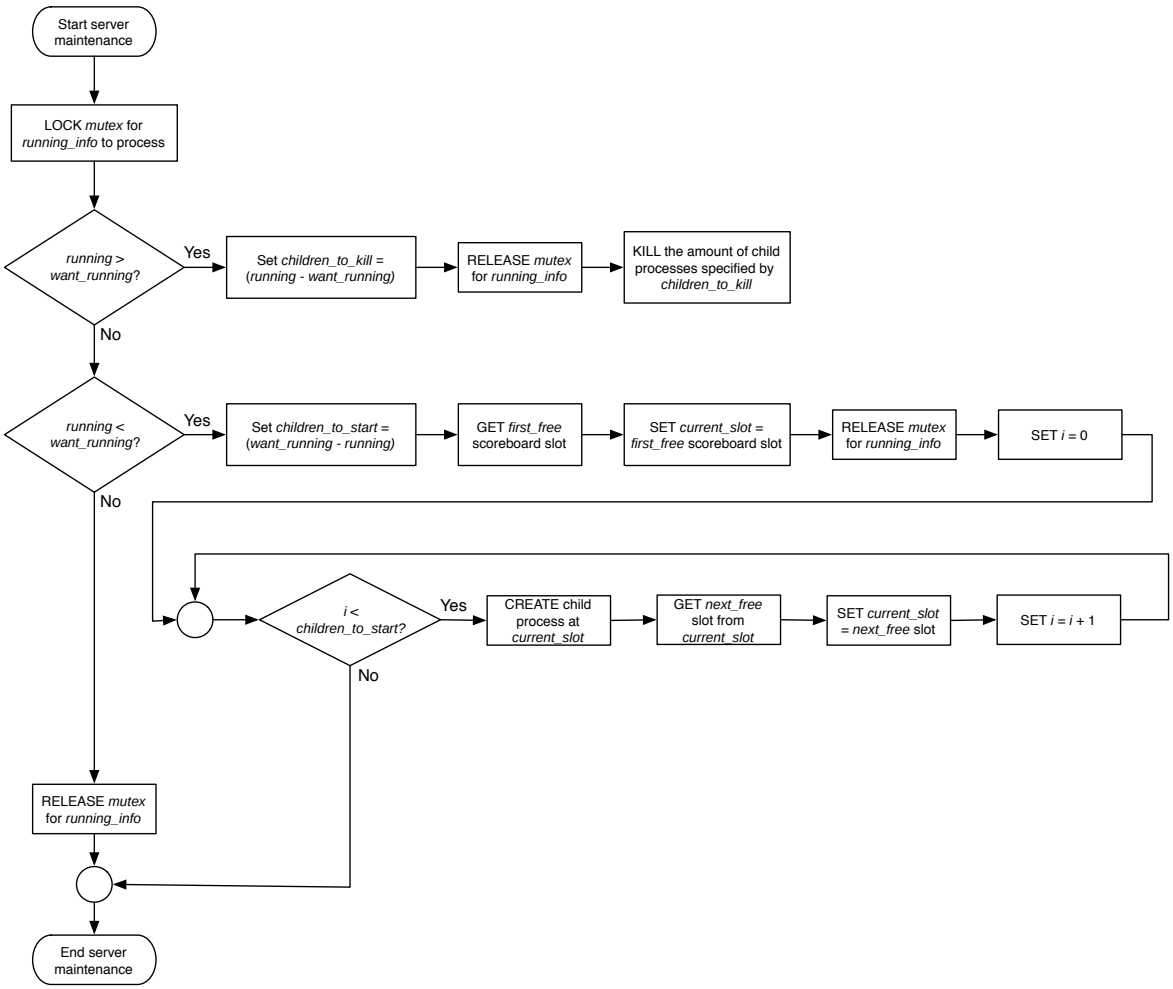

Figure 3.4: A flowchart over the new server maintenance function.

To shutdown child processes Prefork's spare server control uses the *pod* as described in Section 2.2.4 (on page 8). The problem with this approach is that it is just possible to terminate one child process per iteration. With the new child process control it is desirable to be able to kill as many child processes as necessary in one iteration. In Apache there already exists a function that makes this possible with some additions.

The function *ap_mpm_pod_killpg* in *server/mpm_common.c* does not kill a child process on its own, instead it wakes up idle child processes by creating a number of "dummy connections". The amount of child processes to terminate is specified by the parameter *children_to_kill*. The "dummy connection" will simulate a client's connection to the server, where the client immediately closes down its connection. When a listening child process tries to handle a dummy connection, it will discover that the client has disconnected and will therefore not serve the request. The child process will then leave the request-response-loop and enter an area in the code where it examines if it should terminate or keep listening for new requests on the server port. Busy child processes will not be interrupted by the "dummy-connections", they will finish serving their requests before leaving the request-response-loop.

As mentioned earlier the way a child process currently decides if it should terminate or not is to examine the *pod*. With the new approach the parameters *running* and *want_running* should be used for this instead. When *running* is larger than *want_running* the child process should shut down. Because both these parameters are global parameters there will be no problem to use them

from the child processes.

When a child process terminates the *running* parameter needs to be updated, and as mentioned earlier this will be done by the scoreboard function *ap_update_child_status_from_indexes* in the *server/scoreboard.c* file. What has not yet been described is how this should be done with regard to the new parameters *first_free* and *next_free*, because they need to be updated as well. There is no need for keeping the numeric order of the scoreboard slots, so as soon as a slot gets the status *SERVER_DEAD* it can be marked as the *first_free* one and make its *next_free* parameter point to the old *first_free* slot. In Figure 3.5 there is an example of this which shows how the scoreboard setup will change when a child process with slot index 8 has terminated.

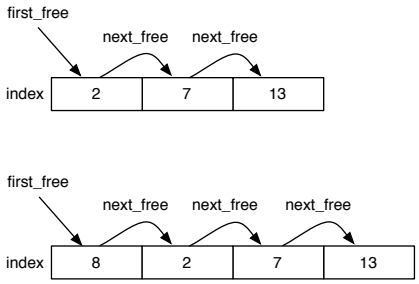

Figure 3.5: The scoreboard setup before and after a child with the slot 8 has terminated.

The slot with index 8 will be inserted as the first free slot in the list of all free scoreboard slots. To link the slot with index 8 to the rest of the list its *next_free* parameter will be made to point to the old *first_free* slot, which was the slot with index 2.

Because of the way the new server maintenance works the child calling *ap_update_child_status_from_indexes* with the status *SERVER_STARTING* will always be the child in the *first_free* slot (you can see this in the flowchart in Figure 3.4). This makes it really easy to update the parameters. In Figure 3.6 there's an example of this which shows how the scoreboard slot setup will change when a child process with slot index 8 starts.

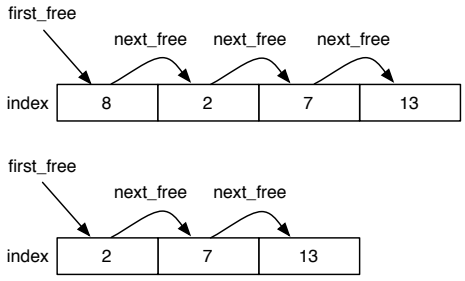

Figure 3.6: The scoreboard setup before and after a new child has been created.

The slot with index 8 will be removed from the list, and its *next_free* slot will be marked as the new *first_slot*. In this case the slot with index 2 will be the new *first_free* slot.

The way the *first_free* and *next_free* parameters will be updated by the *ap_update_child_status_from_indexes* in the *server/scoreboard.c* file is described in the flowchart in Figure 3.7. The flowchart also contains information of how the *running* and *idle_running* parameters will be updated. The actual *C* code for doing this is located in Appendix B.3.

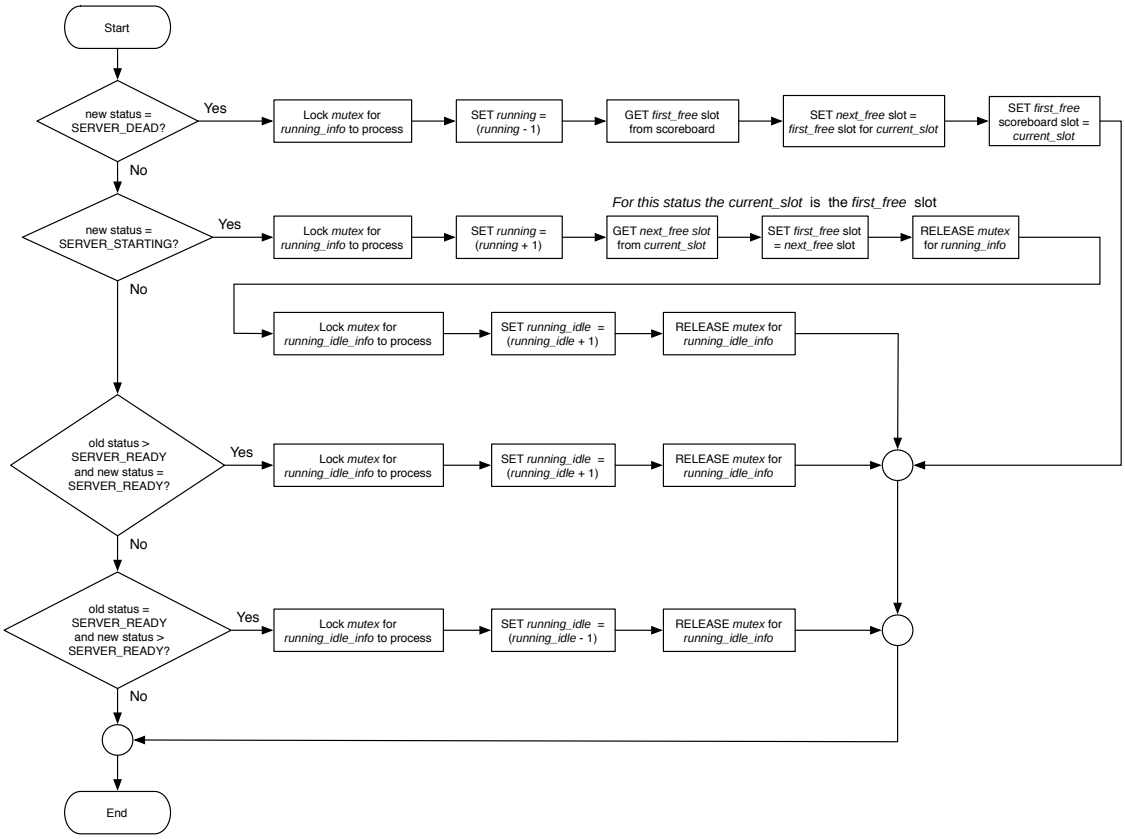

Figure 3.7: A flowchart showing how the four parameters *running*, *idle_running*, *first_free* and *next_free* will be updated from the *ap_update_child_status_from_indexes* function in the scoreboard.

3.3 Influence the HTTP Management

3.3.1 Runtime Update of the KeepAliveTimeout Parameter

In the source code the value of the *KeepAliveTimeout* parameter is stored in a data structure type called *server_rec* which is defined in the file *include/httpd.h*. It contains various information about the server, e.g. the process the server is running on, the name of the server, log file information, module specific information and persistent connection information. In the prefork MPM this data structure is called *ap_server_conf*. To make it possible to change the value of the parameter *KeepAliveTimeout* during runtime, this variable will be put in the *global_score* data structure in the *include/scoreboard.h* file:

```
typedef struct {
    int              server_limit;
    int              thread_limit;
    ap_scoreboard_e  sb_type;
    ap_generation_t  running_generation; /* the generation of children which
                                          * should still be serving requests. */
    apr_time_t  restart_time;
    running_info running_info;
    running_idle_info running_idle_info;
    int want_running
+   apr_interval_time_t keep_alive_timeout;
} global_score;
```

As with the *want_running* parameter the default value of this new global parameter is assigned in Prefork's *ap_mpm_run* function:

```
ap_scoreboard_image->global->keep_alive_timeout = ap_server_conf->keep_alive_timeout;
```

The default value is taken from the value of *KeepAliveTimeout* specified in the configuration file *httpd.conf* which in the code is then stored in the data structure *ap_server_conf*. This data structure is created by the parent process and is made available to all its children because they are all copies of the parent. The reason for not using this data structure is because it is just a copy of the parent one. If the value for the parent structure is changed this change would not go through to the already existing child processes. For child processes created after the change, they will get the updated value. If the value is put in the scoreboard, this will make sure that all processes, no matter if they are already running or not, will eventually get the same value. The *ap_server_conf* is used by the child processes when creating a connection with the client. This is done by calling the function *ap_run_create_connection*. The information in the *ap_server_conf* is among other things used for setting up the persistent connection with the client, so before the child call this function, the value of the *KeepAliveTimeout* needs to be updated from the scoreboard like this:

```
ap_server_conf->keep_alive_timeout = ap_scoreboard_image->global->keep_alive_timeout;
```

3.4 Influence the Network Management

3.4.1 Runtime Update of the ListenBackLog Parameter

To make it possible to update the *ListenBackLog* parameter the same can be done as for the *KeepAliveTimeout* parameter above: putting a new variable in the global data structure of the scoreboard. The problem with this parameter is to make the server pick up a change in it, or more correctly to make the *listen socket* pick up the change.

The value of the *ListenBackLog* directive from the *httpd.conf* file is stored in a variable called *ap_listenbacklog*. This variable is set in the file *server/listen.c*. The value of the parameter *ap_listenbacklog* is just used once when creating the *listen socket* by calling the function *apr_listen* from *server/listen.c*. As described earlier its value is used to define how long the backlog queue of pending connections to the *listen socket* may grow. The *apr_listen* function mentioned above in turn calls the underlying function that the current operating system provides to create a *listen socket*. Under Linux this system function is simply called *listen*. Here is a brief description of how it is possible to work with sockets in Linux.

A socket is first created with the system function *socket*. This function returns a *socket descriptor*, which is an integer value that uniquely identifies a socket. By then calling the *listen* function with this socket descriptor as parameter and the backlog length, the system is notified that this socket should be able to accept incoming connections. It will become a *listen socket*. If the same *socket descriptor* but different backlog value is used once more as parameters to the *listen* function, the max queue length of the backlog queue will be updated. If the queue length is decreased to a value that is less than the current number of pending connections in the queue, these connections will not be removed, but newly incoming connection requests will be blocked. It would be better if the pending connections were removed in case their amount exceed the max length of the backlog queue. Because it is the operating system which specifies how sockets are implemented this functionality needs to be added to the operating system. It is outside the scope of this thesis to actually do this kind of implementation, but a description of how this could be done can be found in Appendix A.1.

3.5 Content Adaptation

There is another way to influence the performance of the Apache HTTP server that have not yet been mentioned, and that is through content adaptation. It is exactly what it sounds like,

a way of adapting the amount of content on a web page depending on the server load. This is often something which is discussed in connection with e.g. news sites; if a critical news story is published on the web and a lot of people try to connect to it at the same time. Instead of not letting people access the page to avoid server overload, a solution is to lessen the quality of the content by e.g. removing all images and remove all dynamic content. This could be done by using several levels of quality. For more information on content adaptation, see [4] [29].

Chapter 4

Measurements

4.1 Introduction

The previous chapter described how to implement certain important server directives in order to be able to update them during runtime. The reason for doing this was to make it possible to influence the performance of the server while it still was running. If these parameters are to be used to create a controller, knowledge about what other impact changes in these parameters have on the server will need to be gathered. In addition there has to be some information about how the amount of traffic to the server affects its performance. Considering this it is important to take measurements of how well the server is performing during execution. But for this to be of any use it must be good measurements and measurements of relevant server data.

The server data of interest can be divided into two levels: *Apache Level* and *Computer Level*.

Apache Level There are several variables in the Apache server itself that would be interesting to measure in order to get some knowledge of how the server reacts to changes. As there is no access to the client's side and will not be possible to measure the end-to-end response time, the *Apache's service time* will be of interest. The *Apache's service time* is the time it takes for the server to handle a single request and is a part of the end-to-end response time. It might give an indication of what is happening with the end-to-end response time. If the request is taken from the *listen socket*'s backlog queue, the time for which this request has spent in the queue will not be a part of its service time. The service time begins when a child process has gotten a request and is about to start processing it. To analyze every single service time value would be far too complex and unnecessary, so instead the average Apache service time for a certain time interval is the variable that will be measured.

Other variables that would be of interest are throughput (number of served requests per second), the number of child processes running, the number of idle child processes and the number of requests.

Computer Level The two most important resources for the server are the CPU and memory of the computer on which the server is running. To measure the CPU and memory load are therefore of interest.

Another resource that is important for the server performance is the backlog queue for the *listen socket* (see Section 2.3.6). Interesting data would be: the size of the queue, how many requests that are in the queue, how long time a request spends in the queue. These kind of measurements will be shown to be quite complicated to accomplish.

Before going into details of how to actually perform the measurements, a description of how to store this data to a file will be made. It is desirable to store this data in a file as *Matlab* arrays. *Matlab* [30] will be used to analyze the data so this will make the measurements easy and flexible to read after they have been performed. The data should be written to the file at a given time interval.

4.2 Logging the Measurements

4.2.1 Module

Would it be possible to create an Apache module (see Section 2.2.7) which is responsible for logging the data? It will have access to the scoreboard functions and it is possible to make it fork a new process that would be responsible for logging the data to file at a given time interval. The problem is that just one instance of the module is desirable, a kind of global module. When this approach was tested it seemed very hard to make this kind of module. Because every child process will have its own instance of the module and therefore its own values of the module parameters (and in this case its own forked process). A lot of child processes would end up logging data to the same file. The main purpose with modules is to handle requests, so to create a module that is just supposed to log data at a given time interval, data which is not really connected to a single request but rather the server as a whole, feels a bit wrong.

4.2.2 Server Process

What really is desirable is a single process dedicated to do the logging and that has access to the scoreboard. In the same way as with the parent process the system call *fork* can be used for doing this. To make sure that the log process has access to the scoreboard the parent process is forked right after it has created the scoreboard. This is done by calling the function *logger_init* at the end of the *ap_create_scoreboard* function in the *server/scoreboard.c* file. It is inside the *logger_init* function that the fork occur. The parent will return to the scoreboard and continue to start the server while the created child process will start doing the logging by calling the function *log_data*. The functions *logger_init*, *log_data* and all other code needed by the logging process is located in the file *server/logger.c*. From now on this process will be called the *logger*. A schematic view of the new server structure is available in Figure 4.1.

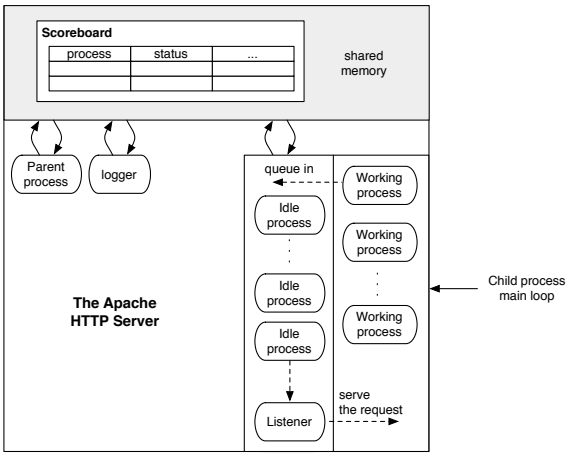

Figure 4.1: A simplified model of the server structure when the *logger* process has been added.

This new process will not be counted as a regular server child process because it has not been created by calling the *make_child* function in the *server/mpm/prefork/prefork.c* file, which means that it has not been assigned a slot in the scoreboard. The server maintenance function will therefore not be affected by the *logger* process.

Periodic logging

It is desirable to be able to specify a time interval at which the *log_data* function should write the data to a file. A first solution to implement a periodic process like this in *C* would be something like this:

```
time_interval = 1 s
LOOP
        Log data
        Wait for time_interval s
END
```

The problem with this approach is that it ignores the time it takes to log the data, which means that the logging will not be performed within the specified time interval but will instead eventually be more and more displaced like the example in Figure 4.2. To take the logging time into account,

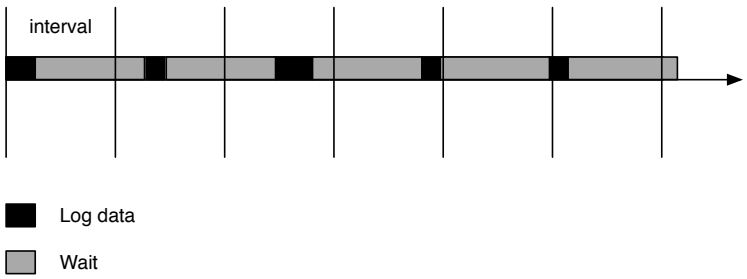

Figure 4.2: Problem with periodic logging.

a start time could be set in the beginning of the loop and a stop time after the logging has been performed. From these two values it is possible calculate how long time the logging took, and subtract that time from the period time. The process will then wait for the remaining time until restarting the loop. Something like this:

```
time_interval = 1 s
LOOP
        SET start_time to current time
        Log data
        SET stop_time to current_time
        SET log_time to stop_time - start_time
        SET wait_time to time_interval - log_time
        Wait for wait_time s
END
```

This is not perfect either, because if the process is switched out just before the *WAIT*, this time will not be taken into account, which means that this solution also might lead to displacement of the specified time interval. But this solution is good enough for my purpose and was the one implemented. To make sure that the time interval is not displaced the current time could be saved in a variable before the loop starts. Then to specify the next time for which the loop should start again the *time_interval* value is added to the loop start time. A process dedicated to store the measurements to a file at a given time interval is now available.

4.3 Measurements on Apache Level

This subsection presents how the measurements for the Apache HTTP Server should be performed. The variables I want to measure in the server are the following:

- throughput (number of requests handled per second),
- Apache average service time (in seconds)
- number of child processes

- number of idle child processes

It is already possible to read information about the number of child processes from the *running_info* struct in the scoreboard (see Section 3.2.3). The next step is to find a way to read information about the *throughput* and *Apache service time*.

4.3.1 Measure Apache Service Time And Throughput

Like the other variables of interest, these are global server data, so it is probably a good idea to store this data in the global section of the scoreboard as well. Instead of storing the actual throughput the number of all finished requests during one sample period (the rate at which the *logger* writes data to file) will be stored. The reason for doing this is that the number of all finished requests might also be an interesting variable to measure. The throughput can then be calculated from the measurements. The same thing will be done for the average service time, i.e. the total service time for all finished requests during one sample period will be stored, which later can be used to calculate the average service time. To get a place to store these two new values the following data structure called *request_info* is added to the *include/scoreboard.h* file:

```
struct request_info {
  int               nbr_of_requests;// total nbr of requests during one sample period
  apr_time_t        requests_time;  // total time it took to process all these requests
  apr_proc_mutex_t  *mutex;         // must be owned to access the above fields
};
```

Functionality to update these two parameters in the struct needs to be added to the scoreboard. A first thought might be to use the *ap_update_child_status_from_indexes* function in the *server/scoreboard.c* file which was used to update the new directives introduced in the previous chapter (see Section 3.2.2). It would be possible to update the *nbr_of_requests* when a child goes from the status *SERVER_BUSY_WRITE* (processing a client request) to the status *SERVER_BUSY_KEEPALIVE* (waiting for more requests from the same client). To obtain the time it took to process a request it is possible to set a start time when a child gets the status *SERVER_BUSY_READ* (reading a client request), set an end time when it gets the status *SERVER_BUSY_KEEPALIVE* and then calculate the process time from those two values. This would certainly work fine, but it is a bit unnecessary when a lot of this functionality already exist. As mentioned in Section 2.2.4 on page 8 the *server* data structure in the scoreboard already contains information about the start- and stop time of a request, so it would be more advantageous to use those times instead.

In the *server/scoreboard.c* file there is a function called: *ap_time_process_request* which is used to measure the time it takes Apache to process a request. It is from this function that the start and stop time information is set. This function is only used if the server has been configured to generate extended status information for a child process. To enable this feature the following line is added to the *httpd.conf* file:

```
ExtendedStatus On
```

When this directive is added a global variable in Apache called *ap_extended_status* is set to 1. This variable is then used in the *ap_process_request* function in the *modules/http/http_request.c* file to decide whether it should call the *ap_time_process_request* function in the scoreboard or not. The *ap_process_request* function, which is part of the http-module is used by the server to process a HTTP request. If the *ap_extended_status* variable is set, it will call the *ap_time_process_request* function in the scoreboard with the status *START_PREQUEST* just before it starts to process the request. When it has finished serving the request it calls the same function but now with the status *STOP_PREQUEST* before leaving the *ap_process_request* function.

When a child process calls the *ap_time_process_request* function with the status *START_PREQUEST* the start time field of the child is set to the current timestamp, and if it has the status *STOP_PREQUEST* the stop time field of the child is set to the current timestamp. Figure 4.3 shows a flowchart of

how to use the the *ap_time_process_request* function to update the parameters in the *request_info* struct. The real code can be found in Appendix B.3.

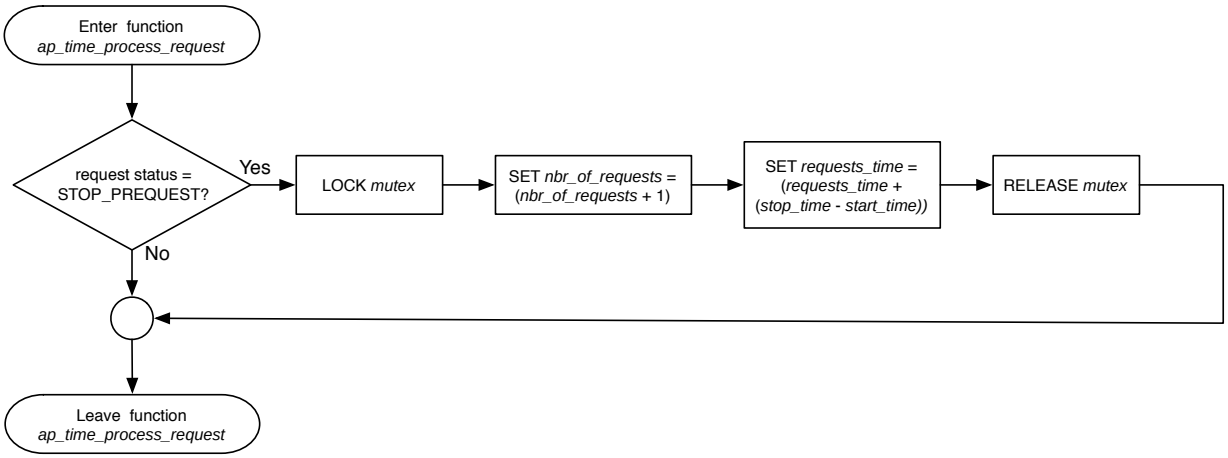

Figure 4.3: A flowchart describing how the two parameters *nbr_of_requests* and *requests_time* in the data structure *control_logdata* should be updated.

4.3.2 Measure Round-Trip Time

The main focus with these measurements is to get a picture of how the server is behaving, but it might also be interesting to get some idea of how busy the network is. This could be done by measuring the round-trip time which is mentioned in Section 2.3.2. A high RTT value should give an indication of a heavy loaded network and a low value a lightly loaded network. From the same place as the requests service time was measured it is possible to add code to measure the RTT value.

Every child process that is processing a request is connected to a *connection socket* and it is from this socket that the RTT value should be obtained. It is possible to do this by using the *getsockopt* system call mentioned in Section 2.3.7. In order to use this function the socket descriptor of the child process' *connection socket* needs to be known. In the server section of the scoreboard there already exists a variable called *sb_socket* which is probably supposed to keep the value of this descriptor, but its value is never assigned from anywhere in the code. So the first thing to do is to add the value of the descriptor to this place in the scoreboard. It is possible to do this by adding the following code to *child_main* function in the *server/mpm/prefork/prefork.c* file, right after the child has accepted a new connection

```
ap_scoreboard_image->servers[my_child_num][0].sb_socket = (apr_socket_t*) csd;
```

The *servers* variable is a pointer to the server section in the scoreboard, *my_child_num* is the current child process' scoreboard slot, 0 is the tread number (because Prefork does not use threads, a thread number of zero will always be used), *csd* is a pointer to the *connection socket* created when the child accepted the request.

4.3.3 Fetch the Measurements

It is now possible to access all the data of interest. To make it easier for the *logger* process and later the controller to get all this data in one go, a function called *ap_get_control_logdata* is added to the *server/scoreboard.c* file. The *logger* will call this function at the interval given by the sample time to fetch the relevant log data.

Before writing this new function the following data structure called *control_logdata* is created. It will hold all of the variables that should be logged:

```
struct control_logdata {
  int          nbr_of_requests;// total nbr of requests during one sample period
  apr_time_t   requests_time;  // total time it took to process all the requests
  int          running;        // nbr of child-processess running
  int          idle_running    // nbr of running child-processess that are idle
};
```

It is the *ap_get_control_logdata* function's task to fill this struct with the right data. It will do so by using the two data structures *running_info* and *request_info*. This function will also reset the two parameters *nbr_of_requests* and *requests_time* in the *request_info* struct to zero after their values have been read, because their values are just valid for one sample period.

4.4 Measurements on Computer Level

4.4.1 Introduction

The variables of interest on the computer level are:

- CPU load
- memory load
- number of requests in the backlog queue

There already exists quite a lot of different utility programs in Linux to get information about both the CPU- and memory load. The most common ones are *top* and *vmstat*. To just get information about the memory usage on the computer it is possible to use the program *free*. There also exists a package to Linux called *sysstat* [31] which contains most of the utilities you need to monitor your system. The main program in this package is called *sar* and it collects, reports and saves system activity information of CPU, memory, disks, interrupts, network interfaces, TTY, kernel tables, etc.

The problem with using these utilities is that they are all command line programs which either display their data in standard output or write their data to a file. This complicates the possibility of using them for my purpose. To implement these measurements myself will make it easier to pass the data to the controller and will also affect the server much less, because to run a command line program from a *C* program at a specified interval is a much heavier task.

4.4.2 How To Measure CPU Load

A common way to measure the CPU load is to use an idle process. An idle process is a process that just runs forever but does not do anything. When the idle process has been created it is assigned the lowest scheduling priority of the operating system. In Linux the command *nice* can be used for this. The priority can be specified in the range of -20 (the highest priority) to 20 (the lowest). When the idle processes has been created it is run for the entire sample period. The CPU load is then calculated by examining for how long time this process was processed by the CPU. This time will be regarded as idle time. For instance if the sample period was 1 s and the idle process was processed for 0.6 s by the CPU it means that 60% of the time the CPU spent doing nothing, which means that the load was 40%. This is the only way to measure the CPU load on a Windows machine, but in Linux there is the proc system (see Section 2.4) which will be used instead.

As mentioned in Section 2.4 the proc system can be used to gather different information about the computer. In order to calculate the CPU usage you can use the *proc* file */proc/stat* (see the following Linux manual page for more info: *man 5 proc*). The first two rows of the file are the important ones for this calculation:

```
cpu  98506 46196 95118 172431403 78531 5007 0 0
cpu0 98506 46196 95118 172431403 78531 5007 0 0
```

If there exists more than one cpu in the computer another line will be displayed after the ones shown above. This new line will start with *cpu1*. The first row contains the total values of all the cpu's, so it is this row that will be used. The different values contain information about the number of jiffies (1/100s of a second) which the cpu has spent performing different actions. From left to right these are as follows [32]:

user normal processes executing in user mode

nice low priority (niced) processes executing in user mode

system processes executing in kernel mode

idle doing nothing

io wait waiting for input/output to a disk device

hard irq

soft irq

steal has to do with running virtual machines

To calculate the CPU load from these values is the same as doing it for the idle process method, but the differences is that the value of the idle time is already known. To get a total time add all these values together. Keep track of how much the total time changes during a sample period and how much the idle time changes during the same period. Calculate the CPU load by comparing how large part of the total change that the idle time change corresponded to.

4.4.3 How To Measure Memory Load

From the */proc/meminfo* (see the following Linux manual page for more info: *man 5 proc*) it is possible to get the following information about the memory usage in kB:

```
MemTotal:       515032 kB
MemFree:         68124 kB
Buffers:         72132 kB
Cached:         282528 kB
```

At first (*MemTotal* - *MemFree*) was used as the value for the total memory usage in kB and I thought that would be fine. But later when the value of the free memory parameter was examined using the command top, I noticed that even though no heavy application was running on the computer, the amount of free memory kept shrinking quite a lot. After reading a faq [33] about how Linux uses its memory I realized that if the memory is not needed for anything else the system uses as much as it can of it to cache files. If an application needs memory and the amount of free memory is not enough some of the cached memory will be released at once. This means that if I wanted measurements of the memory usage that made any sense I needed to treat the cache and the buffers as free memory as well. By trying the command *free* in a console window an extra line with this information is printed:

```
 free
             total       used       free     shared    buffers     cached
Mem:        515032     446908      68124          0      72132     282528
-/+ buffers/cache:      92248     422784
Swap:       987956        120     987836
```

Where used: 92248 = total: 515032 - (free: 68124 + buffers: 72132 + cached: 282528)

To the *logger* a function called *calc_mem_load* is added that will use the */proc/meminfo* file to calculate the used memory in kB. The *C* code for this function is available in Appendix B.6.

4.4.4 How To Measure the Length of the Backlog Queue

It has already been mentioned how the command line tool *netstat* can be used to see which TCP connections that are in the backlog queue or not (see Section 2.3.7 on page 18). If the server is running on port 8000 the following could be written in the command line to get the number of established connections to the server that are currently in the backlog queue:

```
netstat -tp | grep -c ':8000.*\\ ESTABLISHED\\ *-'
```

This line first finds all TCP connections by calling *netstat*. This data is then sent (by using a *pipe*) to the *grep* command which finds out how many of these connections that are connected to port 8000, have the status *ESTABLISHED* and are not yet connected to any process. The problem with using this command to log data e.g. each second is that it is quite a heavy command to run, and it will affect the server's performance negatively. One more downside with this method is that the more connections there are to the server, the longer time the command will take to execute. A good measurement should not affect the server differently depending on the load. A test to run this command from the *logger* was made by using the function *system* which makes it possible in a C-program to call command line programs. But unfortunately this test was never successful.

It is the same problem as with the CPU and memory load when using command line tools. It will be a bit difficult to pass the data to the controller.

After studying the behavior of the */proc/net/sockstat* file I came up with a possible solution. The output from this file looks like this:

```
sockets: used 307
TCP: inuse 10 orphan 0 tw 4 alloc 12 mem 1
UDP: inuse 16
RAW: inuse 0
FRAG: inuse 0 memory 0
```

The interesting line is the one starting with TCP. I have figured out how the following three field work:

orphan shows how many of the TCP sockets that are in the state *FIN_WAIT2*

tw shows how many of the TCP sockets that are in the state *TIME_WAIT*

alloc shows how many sockets that are either in state *LISTEN* or *ESTABLISHED*. It does not matter if a socket with the state *ESTABLISHED* is connected to a process or is in the backlog queue, it will still count towards this field.

From this description it is quite clear that it is the *alloc* field that is of interest when trying to measure the backlog queue length.

If the value of the *alloc* field is stored at the beginning of our measurements, it is possible to see how much it grows. This value will be stored in a variable named *alloc_at_start*. For each new connection to the server its value will increase with 1. The amount it grows to will be equal to the sum of all active child processes (child processes which are processing a request) plus the connections waiting to be processed in the backlog queue. It is possible to calculate the amount of active child processes from the parameters *running* and *running_idle* by taking the difference between those values. So to get the number of connection in the backlog queue the following calculation may be performed:

```
connections in backlog queue = (alloc - alloc_at_start) - (running - running_idle)
```

It is not possible to guarantee that this will correspond to the true amount of connections in the backlog queue for the server, but it is a quite good estimate. The reason why it is not perfect is that the values in the */proc/net/sockstat* file are not connected to the Apache server alone, but to the whole computer, so e.g. if someone logs in to the computer by using *ssh* the value of the *alloc*

field will also grow. But assuming that the main purpose of the computer is to run a web server this estimate is good enough. It gives an idea of what is happening with the queue.

To be able to get the true amount of connections in the backlog queue, functionality needs to be added to Linux's implementation of sockets. How this could be done is explained in Appendix A.1.1.

To the *logger* process a function called *get_sockstat* is added which will get the values for the fields *orphan*, *tw* and *alloc*. These values will then be logged to separate files every sample period. The *C* code for the *get_sockstat* function is available in Appendix B.6. From the logged *alloc* field the number of connections in the backlog queue is then calculated in matlab.

Listen Backlog Queue Overflow

One variable that is available from the *proc* system that would be interesting to measure in connection with the backlog queue length is *ListenOverflows*. It shows the number of times that a *listen socket's* backlog queue has overflown. It is available in the file */proc/net/netstat*. To the *logger* a function called *get_netstat* is added to get information from this file.

Chapter 5

Load testing

5.1 Introduction

In the previous chapter it was shown how some of the most important performance metrics connected to the Apache HTTP server can be measured. With this knowledge it is now time to test the server to find out how the changeable configuration directives implemented in Chapter 3 affect the server's performance. As mentioned earlier to do this there has to be some traffic load on the system and the web server also needs to provide some content which the generated traffic may request. How the server will react to changes in the configuration parameter is highly dependent on the traffic load and the content, so the results presented in this chapter are specific to the chosen setup. The measurements performed in this chapter are mainly done to make sure that the *logger* and the test automation (see Section 5.2.1) work as expected.

5.2 Traffic Generator

There exists quite a lot of traffic generator softwares e.g. *ab*, *httperf*, *JMeter*, *SURGE*, *s-client*. A short description of the most common ones are available in [34].

The *JMeter* traffic generator [35] was chosen because it is an open source project and is well suited for simulating heavy concurrent loads. It is created by the Apache Software Foundation and is written in Java. The code structure of *JMeter* makes it quite easy to extend when additional functionality is needed.

Note however that *JMeter* is not suitable for generating web traffic that should follow a certain statistical distribution. When generating representative workloads this is an important aspect, because such a workload is usually built up by using different distributions. The reason why this is hard to do in *JMeter* is because there is no way of synchronizing the threads to make them send requests at previously specified times. Each thread in *JMeter* runs independently of one another. The generation of representative workload is a complex task which I will not try to achieve for my experiments, because this is not the main goal with this thesis. If you are interested in this subject you can have a look at the references available in Section 2.6 on page 20.

Even if the traffic generated is representative the experiment will not be so if the server settings are unrealistic. For examples most servers today process requests by using persistent connections, which is the default behavior for the Apache server. This is one more reason why the *JMeter* traffic generator was chosen, because it has good support for creating *KeepAlive* HTTP request.

In the experiments it is desirable to test what impact the *KeepAliveTimeout* parameter together with the *want_running* parameter have on the backlog queue length, the *Apache service time*, the end-to-end response time and the round-trip time. To make the experiments more interesting plain deterministic traffic will not be used. Instead each *JMeter* thread will send requests to the server where the intervals between requests will follow the exponential distribution. I do not think it is realistic to put a heavy load on the server by letting each client thread send requests to the

server using very short intervals unless the server is not under a Denial of Service attack (DoS), because then the intervals between request will very likely be short. From my own experience as a web developer what usually puts a high load on a server are among other things if a large amount of clients connect to the server at the same time or if the number of hits for a heavy dynamic page suddenly increases. To use very short intervals between a client's requests will not make the *KeepAliveTimeout* parameter very interesting, because the persistent connection will never time out. Therefore the mean value of the request intervals will be kept quite high and instead load the server more by increasing the number of client threads. Three values of the *KeepAliveTimeout* parameter will be tested: 5, 10 and 15 seconds. To make it possible for a client thread to time out for any of these values the mean interval time will be kept at 5 seconds. The request intervals are created by using *Matlab*. Figure 5.1 shows an example of how the generated interval times look like for one client thread.

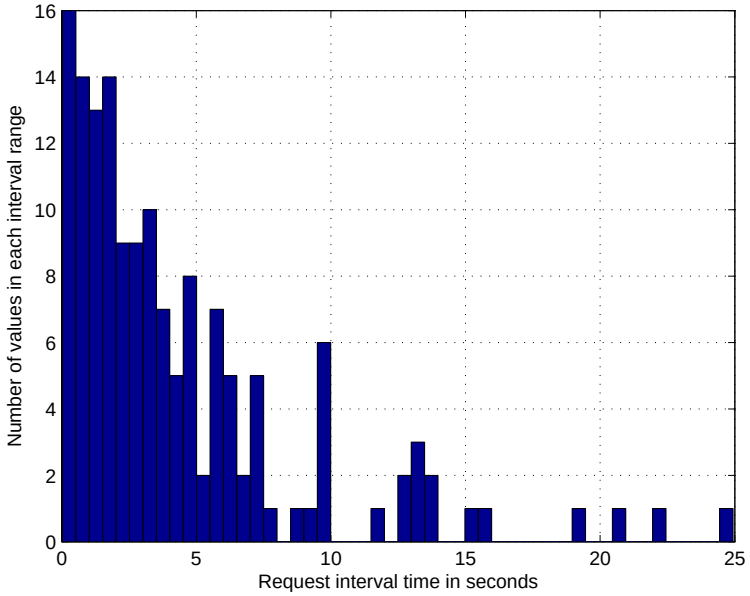

Figure 5.1: The generated interval times for one client thread.

To make it possible for *JMeter* to read these values from file and assign each thread with the right values functionality needs to be added. This is done by creating a new HTTP sampler called *HTTP Interval Sampler*.

5.2.1 The HTTP Interval Sampler

The *HTTP Inter Sampler* is based on the *HTTP Request HTTPClient* sampler in *JMeter*. The reason for extending this sampler was because it was the only one which handled persistent connections as expected (i.e. when using *KeepAlive*). The *HTTP Interval Sampler* extends it by reading generated request interval times, *want_running* values and *rounds* values from files. Each client *JMeter* thread is assigned its corresponding request interval times and *rounds* values (see Section 5.3 on the next page). These threads will use the *HTTP Interval Sampler* to send their requests to the server and the sampler will make sure that the threads wait for the specified interval times before sending a new request.

It is possible to specify how many times each value of the *want_running* parameter should be tested and how many *JMeter* threads which should be used in each one of these tests. When the sampler has reached the end of the last test for a specific *want_running* value it will stop the server's *logger* process from logging more test data to file. As soon as the sampler has loaded a new *want_running* value and setup the threads for its first test for this new value, it will make the server

change its number of running child processes to make it agree with the updated *want_running* value and let the *logger* start writing data to file again. To make it possible for the sampler to send this information to the server and therefore automate the tests, the following three actions have been added to the server:

dummy.php?wr=60&test=1 This will let the server know that a new test is starting, and the *logger* makes sure that 60 child processes will be running before it starts logging data to file. *test=1* means it is the first test and is used by the *logger* to split the log data for each test into separate folders. The *dummy.php* file is an empty PHP script just used for sending these arguments to the server. The reason why using a PHP script for this and not an HTML page, is that an HTML page might be cached by the client. When this happens and new request to the same HTML page arrives, this request will not be processed by Apache, but instead just returned from the cache. The arguments sent with the request will therefore not be processed. This will not happen with a PHP script.

dummy.php?stoptest This will make the *logger* stop writing data to file.

dummy.php?stopwholetest This will make the *logger* stop writing data to file and also terminate the *logger* process.

The functionality for these three actions was added to the *ap_update_child_status_from_indexes* function in the *server/scoreboard.c* file. When a child gets the status *SERVER_BUSY_WRITE* the server will examine the arguments which was sent together with the request the child is about to handle. If it finds any of the three actions above, these will be performed. To synchronize the *logger* i.e. to make it start and stop depending on these actions a *mutex* is used. When the server receives the *stoptest* action, it will lock this *mutex* which will make the *logger* stop running. The *logger* will wait until this *mutex* is released and this happens when a request for a new test is sent to the server. The *C* code for this is available in Appendix B.3.

5.3 Requested Content

Many sites today use scripting languages like e.g. PHP, ASP, JSP and Ruby on Rails to generate dynamic content instead of using static html files. It is also very common that a database is used in connection with generating these dynamic pages. Here is one of the most popular server setups: Linux + Apache HTTP server + MySQL + PHP (where MySQL is a database server using SQL). This setup is usually referred to as *LAMP* [36] and is the setup which will be used for the experiments.

The generated traffic will request a PHP-script called *index.php*. This script will query a table in the *MySQL* database for product names and return these names to the client. To the script it is possible to send a *GET* parameter called *rounds*. This parameter limits the number of results which should be read from the database query. For example to just make the script return a single product name the following request may be sent to the server:

```
index.php?rounds=1
```

The *rounds* parameter is valid in the range 1-3000 and as a result the requested content may vary in size from 0.02 kB to 63.25 kB. The content size is linearly dependent on the *rounds* parameter.

Figure 5.2 shows the time it took for the script to finish depending on the *rounds* parameter. This time was measured on the server side. The graphs shows that the execution time of the *index.php* file increases linearly depending on the *rounds* parameter. A first order model was found for the execution time by using the *matlab* command *polyfit*. This model will match the measured data in a least squares sense.

$$execution\ time = 0.000043 \cdot rounds + 0.0059$$

In relation with the generation of representative workload not much has been written about dynamic generated server content. This is a bit odd because today the use of dynamic content is

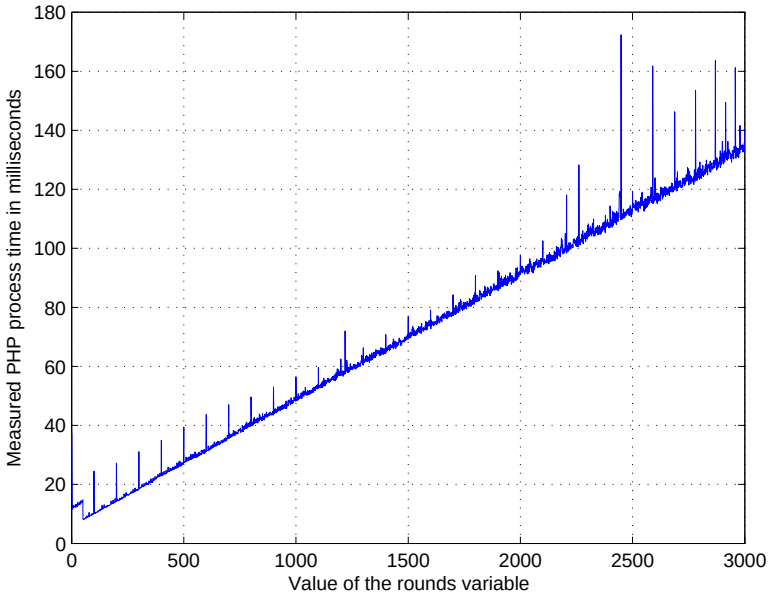

Figure 5.2: Time to execute the index.php script depending on the *rounds* value.

very common and its impact on the server is far greater than static content. The type of content which is requested by the generated traffic is very important in order to get a realistic server load.

Each *JMeter* thread will be made to send *rounds* values to the server which will make the execution times for the *index.php* script follow the Pareto distribution. This means that most of the requests to the server will be served quickly but a small amount of them might take quite a long time to process. As the content size generated by the script is also linearly dependent on the *rounds* parameter, the same can be said about content size. Most of the requested content will be very small, but a small amount of them will be much larger. This assumption might not be so unrealistic when considering the most relevant workload results presented in [25] stating that 10% of the documents account for 90-100% of all requests and bytes transferred, file sizes follow the Pareto distribution and file-interval times are independent and exponentially distributed. These results are based on a survey that was made in 1996, and much has change since then. But these results might still have some validity today, and could be applied to dynamic content.

The Pareto distributed execution time values were generated in *Matlab* and to get the *rounds* values corresponding to these execution times, the inverse of this model for the execution time was used:

$$rounds = \frac{execution\ time - 0.0059}{0.000043}$$

Figure 5.3 on page 46 shows the *rounds* values generated for one *JMeter* thread.

5.4 Experimental Setup

One computer was used to represent the clients by running the *HTTP Interval Sampler* in *JMeter* version 2.2. Five different values of the *want_running* parameter were tested : 60, 120, 180, 240 and 300. For each one of these values seven different traffic loads were generated by running different amounts of *JMeter* threads: 50, 100, 150, 200, 250, 300 and 350. This computer had 512 Mb of RAM and an Intel Pentium 4 CPU 2.4 GHz and used the Feodora Linux Core 5 operating system.

The computer representing the server was running the Apache HTTP server version 2.0.54 with the modifications discussed in the previous chapters. This computer had 512 Mb of RAM and an

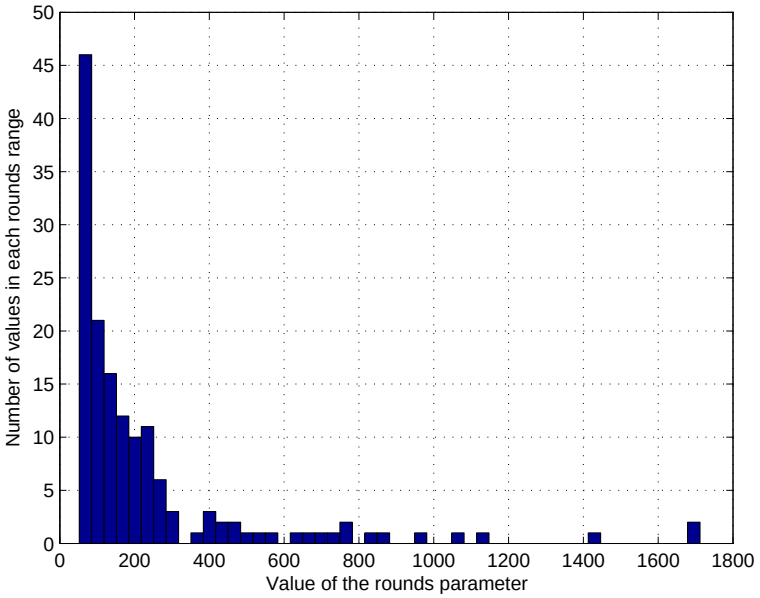

Figure 5.3: The generated *rounds* values for one client thread.

Intel Pentiun 4 CPU 2 GHz and used the Feodora Linux Core 4 operating system.

These two computers were connected through a 100Mbps Ethernet network. A schematic view of the experimental setup is available in Figure 5.4.

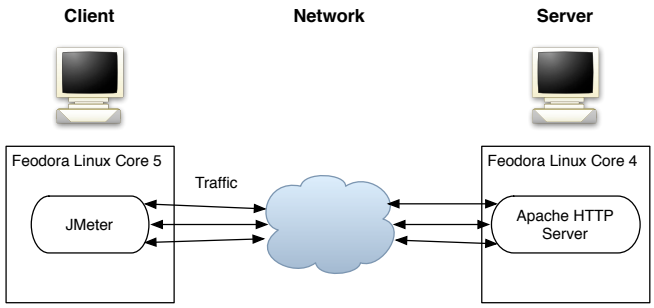

Figure 5.4: The experimental setup.

5.5 Test Results

5.5.1 Introduction

As mentioned previously the *HTTP Interval Sampler* is based on another sampler called *HTTP Request HTTPClient*. The *HTTP Request HTTPClient* sampler waits for a response to its request before it is possible to send a new one. If a lot of requests end up in the backlog queue and still have not been served when it is supposed to send the next request, this will lead to a decrease in the amount of generated traffic to the server. This will be visible in the test results. When this does not happen the average generated traffic for each 50 clients is about 10 requests/second.

5.5.2 Number of Pending Connections In the Backlog Queue

KeepAliveTimeout 5 seconds

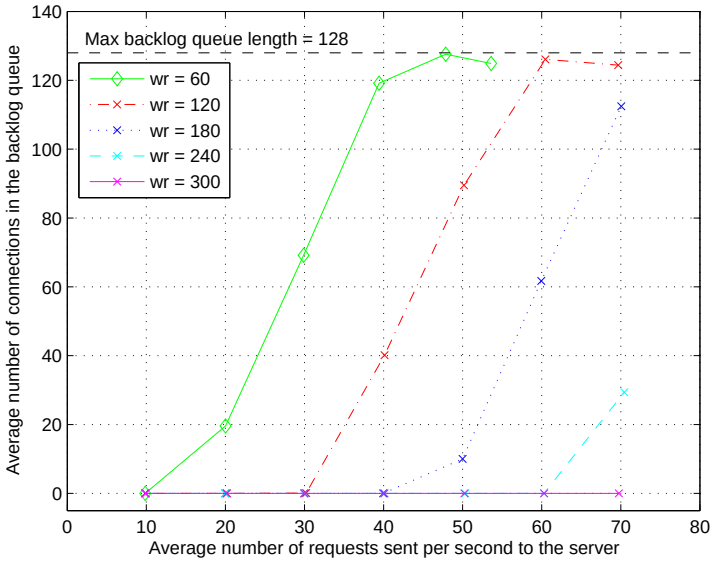

Figure 5.5: Average number of pending connections in the *listen socket's* backlog queue for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 5 seconds.

Figure 5.5 shows that for a short *KeepAliveTimeout* value the time a request spends in the backlog queue will be less and it will require a higher amount of traffic to fill the queue to its maximum. As mentioned before the default maximum queue length for the computer was 128, and Figure 5.5 shows that it is only when 60 or 120 number of child processes are used that this limit is reached. When running the server with 300 child processes the average number of connections in the queue is always zero. The highest traffic, about 70 requests / second, was generated with 350 *JMeter* client threads and even though the number of clients are 50 more than the number of child processes for this traffic the queue does not start to grow. With this short *KeepAliveTimeout* value each client will not keep a single child process busy for that long, so during the sample period (which was 1 second) the probability is quite high that some child process will time out and be ready to accept a new request from the queue.The turn-around time will be less, so each child process will handle requests from more different clients. This should load the server more, which is visible in the *Apache service time* results.

The observant reader will notice that there is only 6 marked points in the graph when using 60 child processes (i.e. $wr = 60$). The reason for this is, as mentioned in the introduction, because it is not possible to generate a higher amount of traffic to the server when it is running with 60 child processes, because many of the requests will get stuck in the backlog queue.

***KeepAliveTimeout* 10 seconds**

Figure 5.6 shows that when the *KeepAliveTimeout* value is increased the queue fills up faster and for a lesser amount of traffic. Now the max limit of 128 connections in the queue is also reached when running 180 child processes in the server. Even when running with 300 child processes some requests will now end up in the queue. The amount of traffic which is possible to generate for the lower *want_running* values is less, because more of the connections will get stuck in the queue faster.

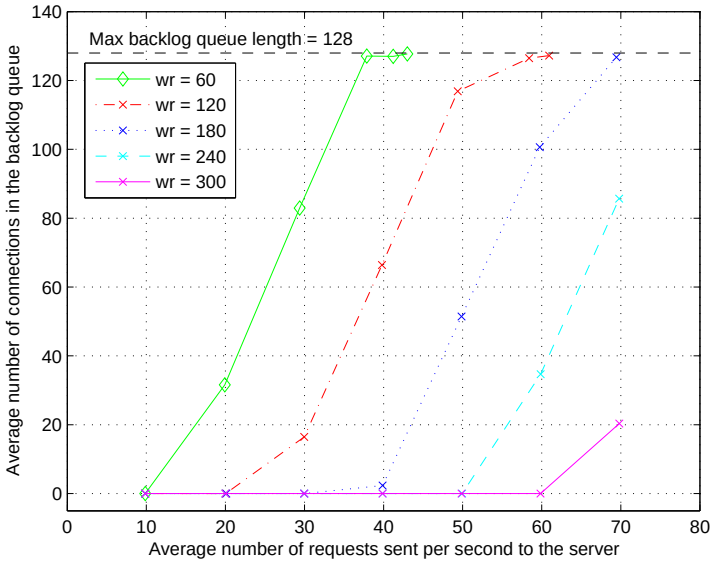

Figure 5.6: Average number of pending connections in the *listen socket's* backlog queue for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 10 seconds.

KeepAliveTimeout 15 seconds

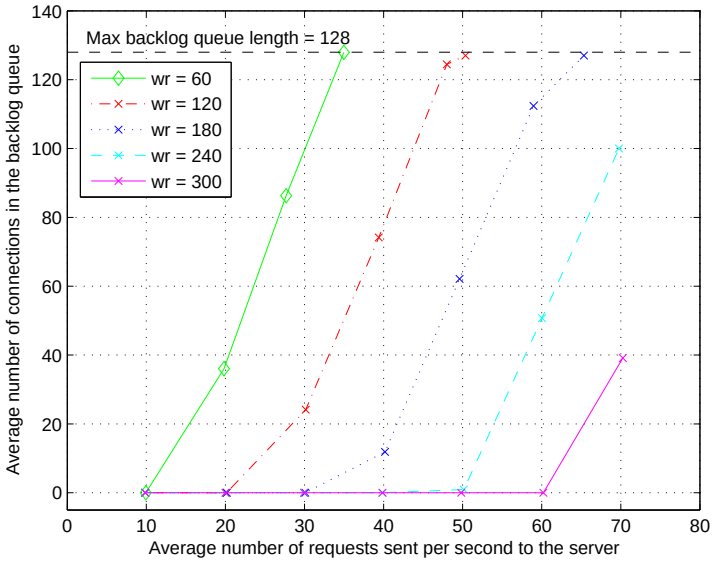

Figure 5.7: Average number of pending connections in the *listen socket's* backlog queue for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 15 seconds.

Figure 5.7 shows the number of requests in the backlog queue for the highest value of *KeepAlive-Timeout* which was tested. For this value the queue fills up faster than before. The amount in the queue is now closer to the difference between the number of clients and child processes. E.g. when

running the server with 300 child processes and a generated traffic with 350 client threads the number of pending connections in the queue is now closer to 50. If the value of *KeepAliveTimeout* was increased even further this value would probably be even closer to 50. The amount of traffic possible to generate is now less even for 180 child processes.

5.5.3 Apache Service Time

***KeepAliveTimeout* 5 seconds**

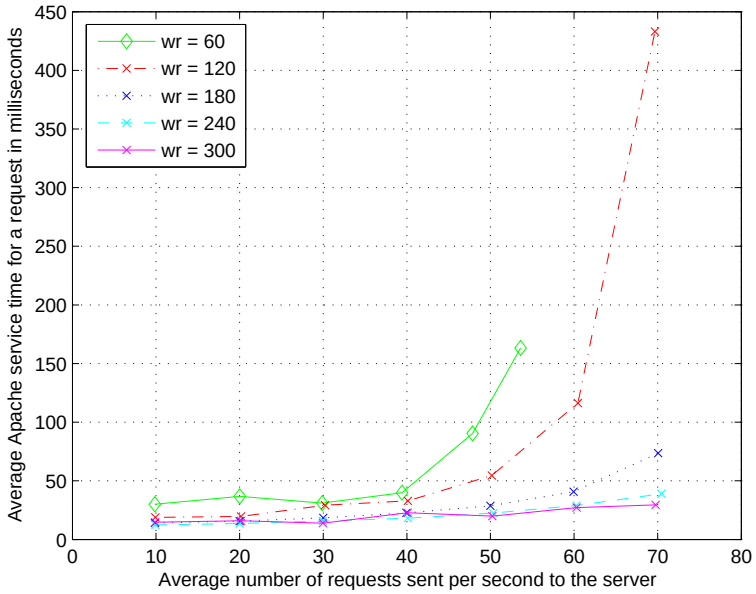

Figure 5.8: Average *Apache service time* for a request in milliseconds for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 5 seconds.

For all of the different number of child processes used the average *Apache service time* is about 20-30 ms for traffic under 40 requests/second. This is visible in Figure 5.8. At 50 requests/second the service time when running 60 or 120 child processes goes up quite much. This seems to happen at the same time as the number of requests in the backlog queue reaches a value around 100 (have a look at the backlog queue graph in Figure 5.5 for *KeepAliveTimeout* 5 seconds). A large number of pending connections in the backlog queue does not seem to be very good for the service time and especially not when the backlog queue overflows. This is most visible when running the server with 120 child processes, because the service time almost reaches 450 ms when the queue overflows, which is about 10 times longer than for lighter traffic.

The result also indicates that running with 300 child processes gives the lowest service time for all the tested traffic loads. When comparing this graph with the backlog queue graph the reason for this seems to be because running with 300 child processes gives the least number of connections in the backlog queue for all the tested traffic loads.

***KeepAliveTimeout* 10 seconds**

For this value of *KeepAliveTimeout* it is also visible in Figure 5.9 that an increase in the service time happens at the same time as the number of connections in the backlog queue increases. The service time increases a small bit for every increase in traffic. This is probably because an increase in traffic means that more requests enter the system which will utilize the CPU and memory more. The most visible change still happens when the queue starts to grow. Because the backlog queue grows faster for the *KeepAliveTimeout* value the differences in service times depending on the

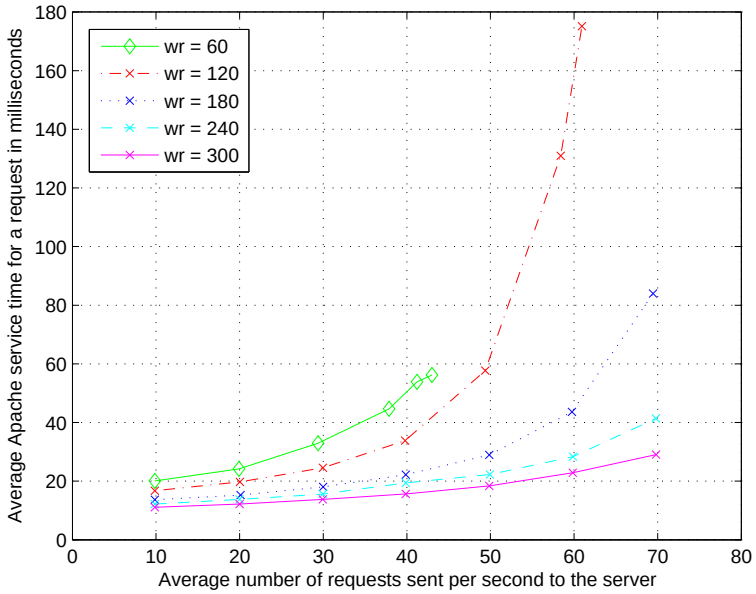

Figure 5.9: Average *Apache service time* for a request in milliseconds for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 10 seconds.

number of child processes are more evident. The service times have also decreased a bit with this value of *KeepAliveTimeout*. The reason for this is probably because more of the served requests have been processed over the same connection because of the increased *KeepAliveTimeout*.

KeepAliveTimeout 15 seconds

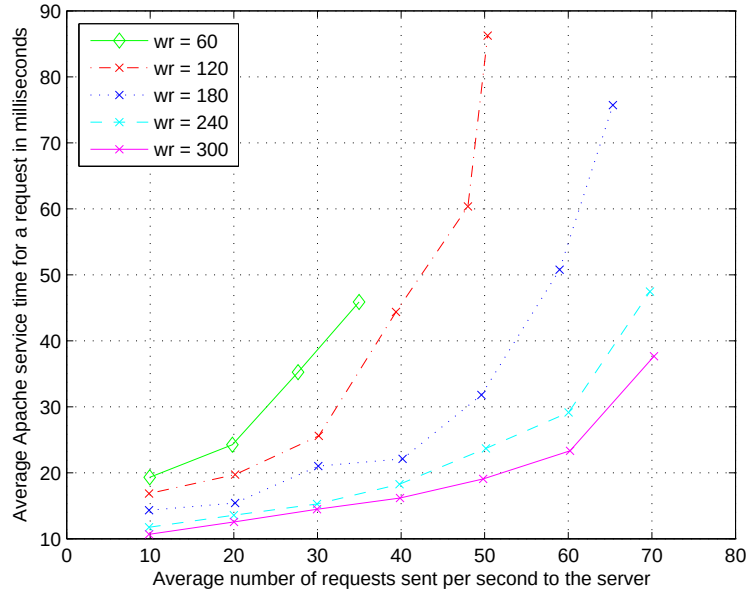

Figure 5.10: Average *Apache service time* for a request in milliseconds for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 15 seconds.

For the highest test value of *KeepAliveTimeout* the same pattern as describe previously occur.

This is visible in Figure 5.10. The service times for all the different numbers of child processes increase a bit when the traffic increases and when the backlog queue starts to grow the service time increases a lot more than before. Before this happens the service times are lower than for the two other values of *KeepAliveTimeout*. The reason for this is the same as mentioned before. There was one thing that I noticed in this graph that I did not see in the other two. As mentioned above when using a higher value of *KeepAliveTimeout* seems to give a lower service time if not the backlog queue starts to grow, but when it starts to grow the graph indicates that the service time will reach a higher value than for a lower *KeepAliveTimeout* value. E.g. look at the end value when running with 240 child processes. In the previous graph it reached a value of just above 40, and in this graph it is more closer to 50. The reason why this is not visible for the lower values of child processes e.g. 60, is probably because it is not possible to generate a higher amount of traffic than 30 request / second. If it was possible to do that it would probably lead to a service time value that was higher than the highest values in the previous graphs.

5.5.4 Round-trip Time

In the Figures 5.11, 5.12 and 5.13 the average estimated round-trip times are displayed for the three different *KeepAliveTimeout* values. It is obvious in all graphs that the round-trip time increases a lot when the backlog queue overflows. For a higher value of the *KeepAliveTimeout* parameter the average round-trip time will be lower even in those cases when the backlog queue overflows. The reason for this is probably the same as with the decrease in the *Apache service time*. If a higher *KeepAliveTimeout* is used, more requests served by the server have been processed over persistent connections, which should lead to a faster response time.

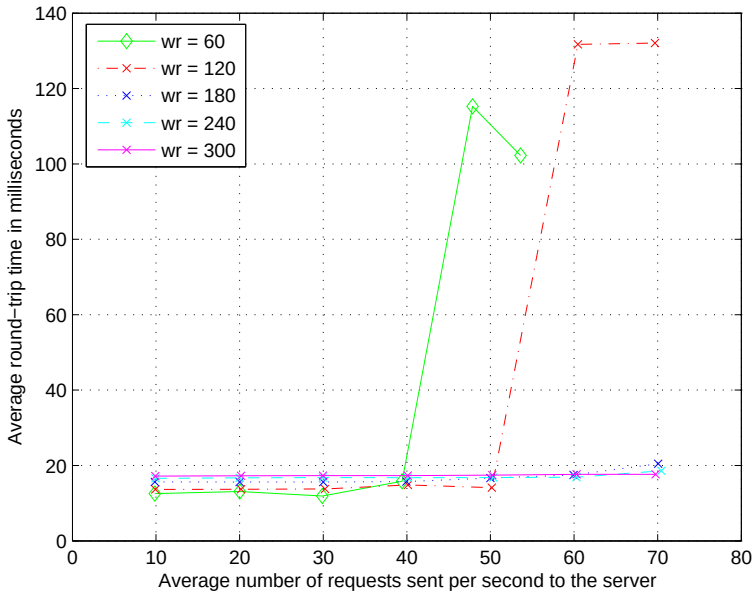

Figure 5.11: Average estimated round-trip time for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 5 seconds.

5.5.5 The End-To-End Response Time

To the *HTTP Interval Sampler* code was added to save the true end-to-end response time for each request sent by the client threads. This data will not be available for use in the controller, but I thought it would be interesting to see how this data is related to the other available parameters. The end-to-end response times for the different values of the *KeepAliveTimeout* parameter are

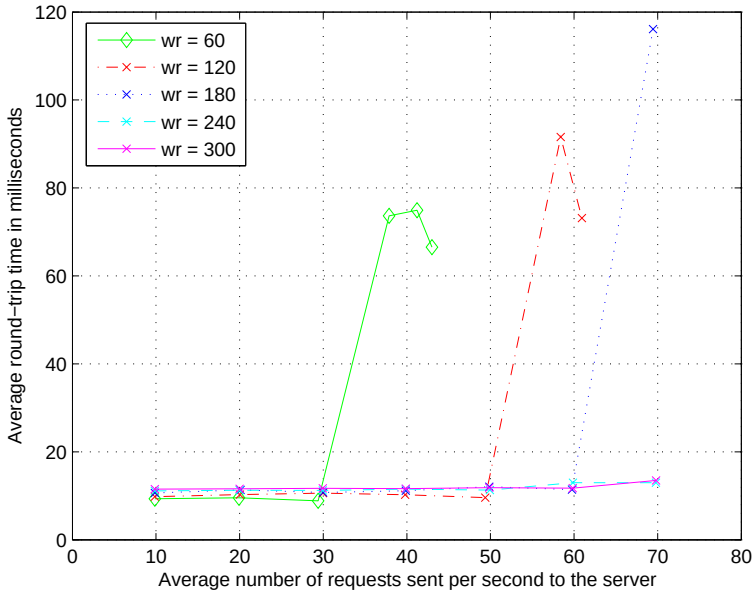

Figure 5.12: Average estimated round-trip time for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 10 seconds.

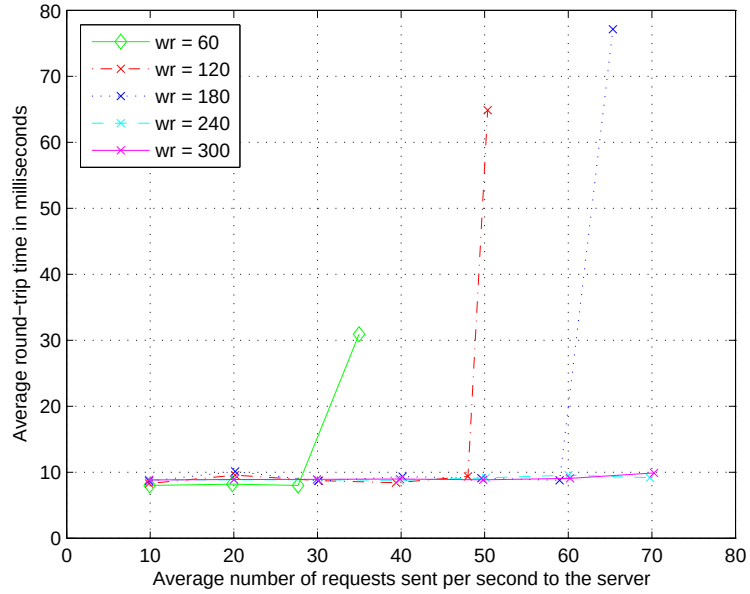

Figure 5.13: Average estimated round-trip time for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 15 seconds.

available in the Figures 5.14, 5.15 and 5.16. In the graphs it is very clear that the response time increases linearly when the backlog queue starts to grow. The backlog queue graphs look very similar to the graphs displaying the response times. This confirms the fact that a queue is problematic when it comes to fast response times.

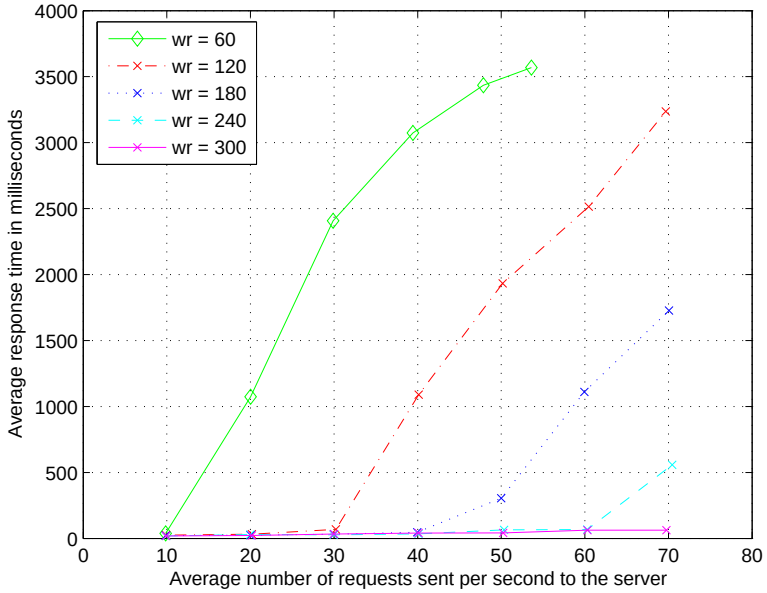

Figure 5.14: Average end-to-end response time for the clients for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 5 seconds.

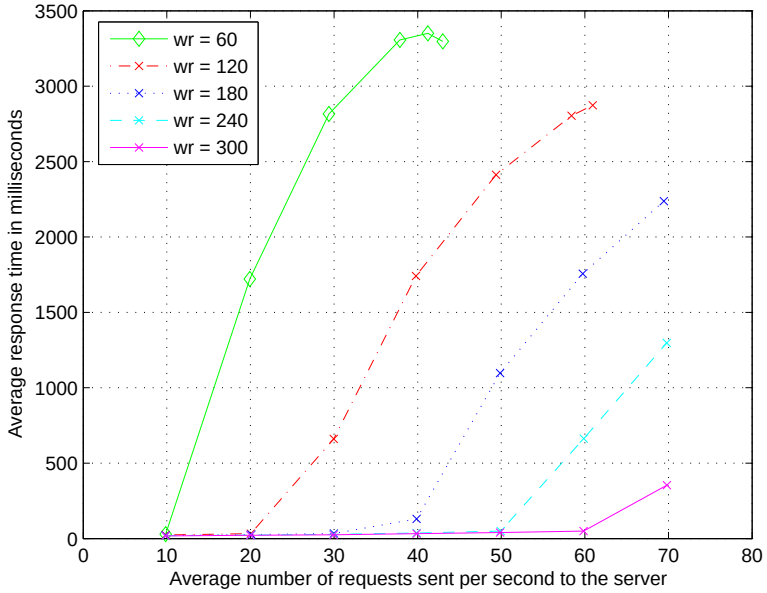

Figure 5.15: Average end-to-end response time for the clients for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 10 seconds.

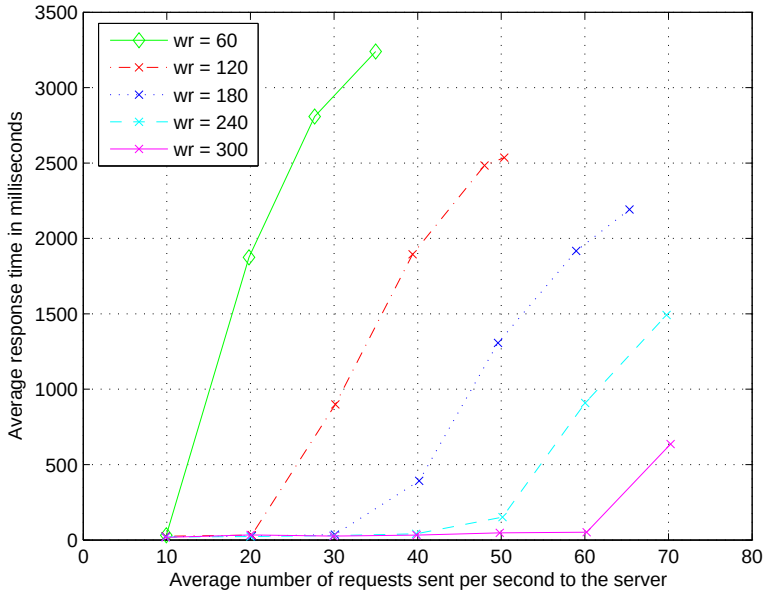

Figure 5.16: Average end-to-end response time for the clients for different values of the *want_running* parameter and traffic load using a *KeepAliveTimeout* value of 15 seconds.

Chapter 6

Controller

The tools needed for affecting and measuring the server's performance are now available to use, and it is time to create a controller. First it has to be decided were it should be placed. It ought to work quite similar to the *logger* process in the sense that it should be a process running periodically and have access to all the functions which the *logger* has. An alternative would be to make it event-based instead of periodical, but for this thesis the periodical way was chosen. When testing the controller it is desirable to have the *logger* running as well, so it will be possible to monitor what the controller actually does.

For the same reasons as for creating the *logger* as a stand-alone process in Apache, the *controller* will be created in the same way and started right after the scoreboard has been initiated. The same technique as for the *logger* will be used to make the *controller* run periodically with a specific time interval. The updated server structure is available in Figure 6.1.

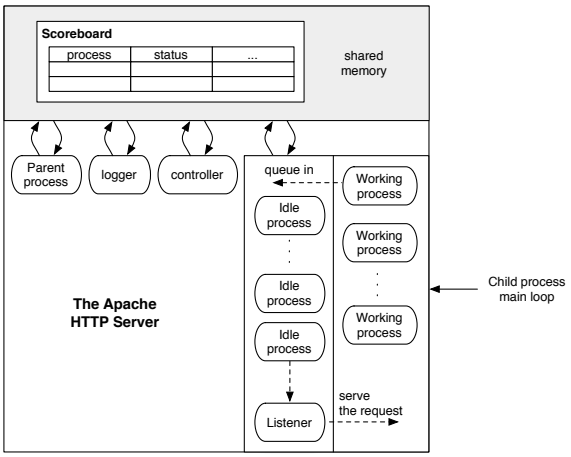

Figure 6.1: A simplified model of the server structure when the *controller* process has been added.

In the previous chapter just one kind of traffic and content were tested and for this setup it was always best to use as high value of the *want_running* parameter as possible. This will probably not be the case for other kinds of workloads. To be able to create a controller that would really improve a real working server's performance, better knowledge needs to be gathered about that particular server's workload. Guidelines of how to do that are available in [25].

To make use of the tools created in the previous chapters a controller will be implemented that will work like Apache's own spare servers control. This means that the number of idle child processes will be regulated so it falls in between a specified minimum and maximum value.

The controller will be tested by using the heaviest generated traffic from the previous chapter

(i.e. by using 350 *JMeter* client threads) and the clients will requests the same content as before.

Figure 6.2 shows how well the controller managed to keep the number of idle child processes in the specified range 32-64. At one point the number of child processes reaches 30 and the reason for this is probably because the *logger* and controller are not yet synchronized. This means that it might be possible for the *logger* to write data to file before the controller has been able to perform its control loop using the same data.

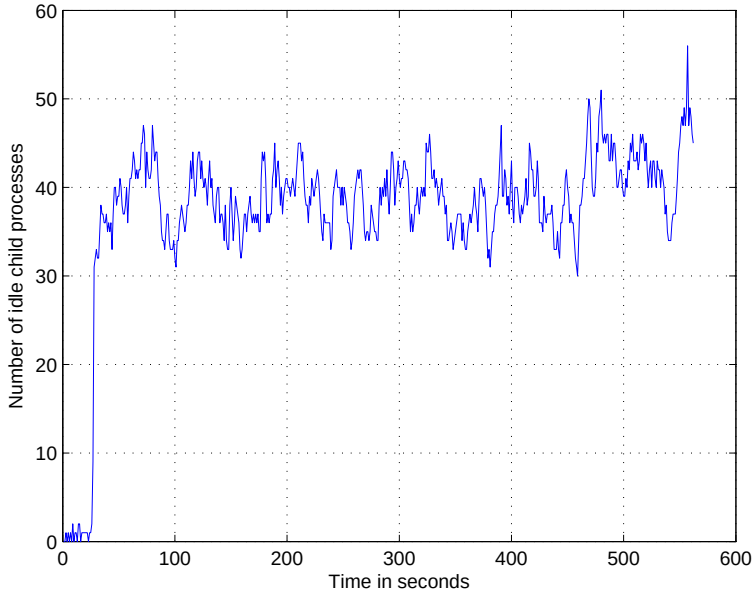

Figure 6.2: Testing the controller which should keep the number of idle child processes in the range 32-64.

Chapter 7

Conclusions and Further Work

To be able to control any type of system one needs to be able to affect the system in some way and measure how it is reacting to the change. In this thesis I have described how to make this possible for the Apache HTTP server by implementing functionality to update important configuration directives and to read information about different areas of the server during run time.

The following API functions have been added successfully to supply the functionality:

ap_set_want_running which is used for specifying the total amount of child processes that should be in the system

ap_set_keep_alive_timeout which is used for specifying the amount of seconds that a persistent connection may be idle before it should be terminated

ap_set_backlog which is used for specifying the maximum amount of pending connections allowed in the *listen socket's* backlog queue

ap_fetch_control_data which is used for collecting all the measured data in Apache

ap_calc_cpu_load which is used to collect various information about the computer's CPU load.

ap_calc_mem_usage which is used for collecting various information about the computer's memory usage

ap_get_netstat which is used for collecting various data from the */proc/net/netstat* file

ap_get_sockstat which is used for collecting various data from the */proc/net/sockstat* file

Support has also been added for doing multiple tests on the server automatically. This functionality was used when *JMeter's HTTP Request HTTPClient* sampler was extended by the *HTTP Interval Sampler*. This sampler was used in Chapter 5 to perform 35 test runs for each of the three different values of the *KeepAliveTimeout* parameter.

In Chapter 6 a shell was created for the *controller* and a simple control algorithm was implemented and tested. The tests performed in Chapters 5 and 6 have shown that the added functionality works as expected.

7.1 Further research

At the moment if something is changed in the *logger* or controller the whole Apache server needs to be recompiled, which is not time efficient. To move away from this way of working, support could be added for changing some default settings for these two processes by using the configuration file. For instance for the *logger* it should be possible to specify what parameters that should be written to file and at what time interval. Both the *logger* and controller should be possible to turn

off. To make it easy to test different control strategies the control algorithms should be possible to load dynamically. As mentioned in Chapter 6, the *logger* and controller are not synchronized, which needs to be supported in order to get good measurements when these processes are running concurrently. The Linux's system calls which I have used in my code should be replaced with their corresponding APR-functions (see Section 2.2.6 on page 10).

Appendix A

Modifications in the Linux Kernel

A.1 Realtime update of the backlog queue

As mentioned in Section 2.3.7 it is possible in Linux to use the system call *setsockopt* to set various options for the socket. An option could be added to this function to change the length of the backlog queue. The functionality for the TCP socket is implemented in the files starting with *tcp* under the folder *net/ipv4* in the Linux source code distribution. In the *net/ipv4/tcp.c* file the *setsockopt* function is implemented for this socket type, and is called *tcp_setsockopt*. To this function the option *TCP_CHANGEBACKLOG* could be added. A way of how this could be implemented is described by the flowchart in Figure A.1 and is also available in *C* code on the next page. Bare in mind that this code has not been tested yet.

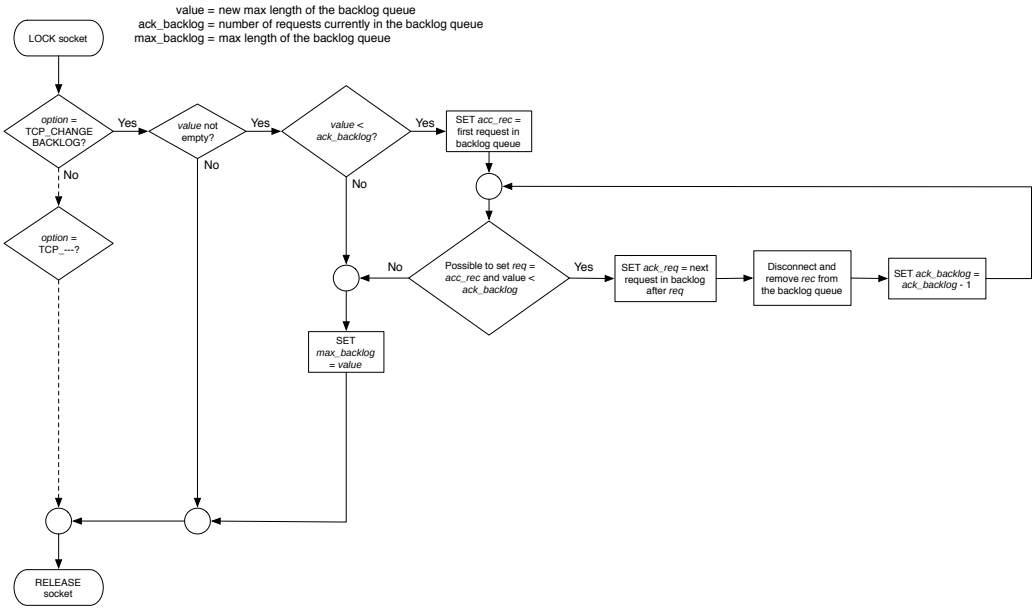

Figure A.1: A flowchart over how the option *TCP_CHANGEBACKLOG* could be implemented.

When this change has been made to the source code, Linux has to be re-compiled.

When everything is up and running, this is what you would have to do in order to update the length of the *listen socket*'s backlog queue. For example if the variable *listener* contains the descriptor to the *listen socket*, in order to change the length of the backlog queue it would be possible to do the following:

```
int option_value = 400;
socklen_t socklen = sizeof(option_value);
setsockopt(listener, SOL_TCP, TCP_CHANGEBACKLOG, &option_value, socklen);
```

The first input parameter to the function is our descriptor. The second is the protocol number for TCP, which is defined by the constant *SOL_TCP*. The protocol number is also available in the file */etc/protocols*. The third parameter is the socket option, which in this case is the option just created. The fourth parameter contains the new value of the queue length, and the last parameter the size of the option value parameter.

A.1.1 Code for the TCP_CHANGEBACKLOG Option

```
1  case TCP_CHANGEBACKLOG:
2      if (val) {
3        if(sk->sk_ack_backlog > val) {
4          struct open_request *acc_req = tp->accept_queue;
5          while ((req = acc_req) != NULL && sk_ack_backlog > val) {
6            struct sock *child = req->sk;
7
8            acc_req = req->dl_next;
9
10           local_bh_disable();
11           bh_lock_sock(child);
12           BUG_TRAP(!sock_owned_by_user(child));
13           sock_hold(child);
14
15           tcp_disconnect(child, O_NONBLOCK);
16
17           sock_orphan(child);
18
19         atomic_inc(&tcp_orphan_count);
20
21         tcp_destroy_sock(child);
22
23         bh_unlock_sock(child);
24         local_bh_enable();
25         sock_put(child);
26
27         sk_acceptq_removed(sk);
28         tcp_openreq_fastfree(req);
29         }
30       }
31       sk->sk_max_backlog = val;
32     break;
```

A.2 Getting the Length of the Backlog Queue

There also exists a system call called *getsockopt* which can be used with a socket to get information about certain parameters. Unfortunately there exists no option to get the number of pending connections in the backlog queue, so if this functionality is desirable it has to be implemented.

The source code for the *getsockopt* function is located in the *net/ipv4/tcp.c* file in the Linux source code. For the TCP socket this function is called *tcp_getsockopt*. To get the desired data from this function an option called e.g. *TCP_GETBACKLOG* could be added with the following code:

```
case TCP_GETBACKLOG:
  val = sk->sk_ack_backlog;
  break;
```

Appendix B

Code

The code which have been added are available in the code blocks starting with *#ifdef EL_CONTROL*.

B.1 prefork.c

```
427 /**********************************************************
428  * Child process main loop.
429  * The following vars are static to avoid getting clobbered by longjmp();
430  * they are really private to child_main.
431  */
432
433 static int requests_this_child;
434 static int num_listensocks = 0;
435 static ap_listen_rec *listensocks;
436
437 int ap_graceful_stop_signalled(void)
438 {
439     /* not ever called anymore... */
440     return 0;
441 }
442
443
444 static void child_main(int child_num_arg)
445 {
446     apr_pool_t *ptrans;
447     apr_allocator_t *allocator;
448     conn_rec *current_conn;
449     apr_status_t status = APR_EINIT;
450     int i;
451     ap_listen_rec *lr;
452     int curr_pollfd, last_pollfd = 0;
453     apr_pollfd_t *pollset;
454     int offset;
455     void *csd;
456     ap_sb_handle_t *sbh;
457     apr_status_t rv;
458     apr_bucket_alloc_t *bucket_alloc;
459
460     mpm_state = AP_MPMQ_STARTING; /* for benefit of any hooks that run as
        this
461                                   * child initializes
```

```
462                                     */
463
464     my_child_num = child_num_arg;
465     ap_my_pid = getpid();
466     csd = NULL;
467     requests_this_child = 0;
468
469     ap_fatal_signal_child_setup(ap_server_conf);
470
471     /* Get a sub context for global allocations in this child, so that
472      * we can have cleanups occur when the child exits.
473      */
474     apr_allocator_create(&allocator);
475     apr_allocator_max_free_set(allocator, ap_max_mem_free);
476     apr_pool_create_ex(&pchild, pconf, NULL, allocator);
477     apr_allocator_owner_set(allocator, pchild);
478
479     apr_pool_create(&ptrans, pchild);
480     apr_pool_tag(ptrans, "transaction");
481
482     /* needs to be done before we switch UIDs so we have permissions */
483     ap_reopen_scoreboard(pchild, NULL, 0);
484     rv = apr_proc_mutex_child_init(&accept_mutex, ap_lock_fname, pchild);
485     if (rv != APR_SUCCESS) {
486         ap_log_error(APLOG_MARK, APLOG_EMERG, rv, ap_server_conf,
487                      "Couldn't initialize cross-process lock in child");
488         clean_child_exit(APEXIT_CHILDFATAL);
489     }
490
491     if (unixd_setup_child()) {
492   clean_child_exit(APEXIT_CHILDFATAL);
493     }
494
495     ap_run_child_init(pchild, ap_server_conf);
496
497     ap_create_sb_handle(&sbh, pchild, my_child_num, 0);
498
499     (void) ap_update_child_status(sbh, SERVER_READY, (request_rec *) NULL);
500
501     /* Set up the pollfd array */
502     listensocks = apr_pcalloc(pchild,
503                               sizeof(*listensocks) * (num_listensocks));
504     for (lr = ap_listeners, i = 0; i < num_listensocks; lr = lr->next, i++)
        {
505         listensocks[i].accept_func = lr->accept_func;
506         listensocks[i].sd = lr->sd;
507 #ifdef EL_CONTROL
508     // trim some socket settings
509         int option_value;
510         socklen_t socklen = sizeof(option_value);
511         option_value = 1;
512         int rsock = setsockopt(lr->sd->socketdes, SOL_TCP, TCP_KEEPIDLE,
            &option_value, socklen);
513         if(rsock < 0) {
514           ap_log_error(APLOG_MARK, APLOG_ERR, 0, ap_server_conf,"Error
              setting TCP_KEEPIDLE to %i: %s", option_value,
              strerror(errno));
515         }
```

```
516             option_value = 1;
517             rsock = setsockopt(lr->sd->socketdes, SOL_TCP, TCP_KEEPINTVL,
                    &option_value, socklen);
518             if(rsock < 0) {
519               ap_log_error(APLOG_MARK, APLOG_ERR, 0, ap_server_conf,"Error
                      setting TCP_KEEPINTVL to %i: %s", option_value,
                      strerror(errno));
520             }
521             option_value = 1;
522             rsock = setsockopt(lr->sd->socketdes, SOL_TCP, TCP_KEEPCNT,
                    &option_value, socklen);
523             if(rsock < 0) {
524               ap_log_error(APLOG_MARK, APLOG_ERR, 0, ap_server_conf,"Error
                      setting TCP_KEEPCNT to %i: %s", option_value,
                      strerror(errno));
525             }
526             option_value = 1;
527             rsock = setsockopt(lr->sd->socketdes, SOL_TCP, TCP_SYNCNT,
                    &option_value, socklen);
528             if(rsock < 0) {
529               ap_log_error(APLOG_MARK, APLOG_ERR, 0, ap_server_conf,"Error
                      setting syncnt to %i: %s", option_value, strerror(errno));
530             }
531 #endif
532         }
533
534         pollset = apr_palloc(pchild, sizeof(*pollset) * num_listensocks);
535         pollset[0].p = pchild;
536         for (i = 0; i < num_listensocks; i++) {
537             pollset[i].desc.s = listensocks[i].sd;
538             pollset[i].desc_type = APR_POLL_SOCKET;
539             pollset[i].reqevents = APR_POLLIN;
540         }
541
542         mpm_state = AP_MPMQ_RUNNING;
543
544         bucket_alloc = apr_bucket_alloc_create(pchild);
545
546         while (!die_now) {
547       /*
548        * (Re)initialize this child to a pre-connection state.
549        */
550
551       current_conn = NULL;
552
553       apr_pool_clear(ptrans);
554
555       if ((ap_max_requests_per_child > 0
556            && requests_this_child++ >= ap_max_requests_per_child)) {
557           clean_child_exit(0);
558       }
559
560       (void) ap_update_child_status(sbh, SERVER_READY, (request_rec *) NULL);
561
562       /*
563        * Wait for an acceptable connection to arrive.
564        */
565
```

```
566     /* Lock around "accept", if necessary */
567     SAFE_ACCEPT(accept_mutex_on());
568
569             if (num_listensocks == 1) {
570                 offset = 0;
571             }
572             else {
573                 /* multiple listening sockets - need to poll */
574         for (;;) {
575                     apr_status_t ret;
576                     apr_int32_t n;
577
578                     ret = apr_poll(pollset, num_listensocks, &n, -1);
579                     if (ret != APR_SUCCESS) {
580                         if (APR_STATUS_IS_EINTR(ret)) {
581                             continue;
582                         }
583                       /* Single Unix documents select as returning errnos
584                        * EBADF, EINTR, and EINVAL... and in none of those
585                        * cases does it make sense to continue.  In fact
586                        * on Linux 2.0.x we seem to end up with EFAULT
587                        * occasionally, and we'd loop forever due to it.
588                        */
589                       ap_log_error(APLOG_MARK, APLOG_ERR, ret, ap_server_conf,
590                                    "apr_poll: (listen)");
591                       clean_child_exit(1);
592                     }
593                     /* find a listener */
594                     curr_pollfd = last_pollfd;
595                     do {
596                         curr_pollfd++;
597                         if (curr_pollfd >= num_listensocks) {
598                             curr_pollfd = 0;
599                         }
600                         /* XXX: Should we check for POLLERR? */
601                         if (pollset[curr_pollfd].rtnevents & APR_POLLIN) {
602                             last_pollfd = curr_pollfd;
603                             offset = curr_pollfd;
604                             goto got_fd;
605                         }
606                     } while (curr_pollfd != last_pollfd);
607
608                     continue;
609                 }
610             }
611       got_fd:
612     /* if we accept() something we don't want to die, so we have to
613      * defer the exit
614      */
615 #ifdef EL_CONTROL
616         ap_server_conf->keep_alive_timeout =
            ap_scoreboard_image->global->keep_alive_timeout;
617 #endif
618             status = listensocks[offset].accept_func(&csd,
619                                                      &listensocks[offset],
                                                            ptrans);
620             SAFE_ACCEPT(accept_mutex_off());  /* unlock after "accept" */
621
```

```
622             if (status == APR_EGENERAL) {
623                 /* resource shortage or should-not-occur occured */
624                 clean_child_exit(1);
625             }
626             else if (status != APR_SUCCESS) {
627                 continue;
628             }
629
630         /*
631          * We now have a connection, so set it up with the appropriate
632          * socket options, file descriptors, and read/write buffers.
633          */
634
635         current_conn = ap_run_create_connection(ptrans, ap_server_conf, csd,
                 my_child_num, sbh, bucket_alloc);
636
637         if (current_conn) {
638     #ifdef ELCONTROL
639             ap_scoreboard_image->servers[my_child_num][0].sb_socket =
                     (apr_socket_t*) csd;
640             ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Keep Alive
                     Timeout (child %i): %i",
                     ap_my_pid, apr_time_sec(ap_server_conf->keep_alive_timeout));
641             ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Scoreboard
                     image Keep alive timeout: %i",
                     apr_time_sec(ap_scoreboard_image->global->keep_alive_timeout));
642     #endif
643             ap_process_connection(current_conn, csd);
644               ap_lingering_close(current_conn);
645               }
646
647     #ifdef ELCONTROL
648         ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Child finished
                 request keep alive timeout exceeded -- running: %i want_running:
                %i",
                ap_scoreboard_image->global->running_info.running, ap_scoreboard_image->global->want_
649         ap_update_child_rounds(sbh);
650         apr_proc_mutex_lock(ap_scoreboard_image->global->running_info.mutex);
651               if(ap_scoreboard_image->global->running_info.running >
                       ap_scoreboard_image->global->want_running) {
652                 die_now = 1;
653               } else if (ap_my_generation !=
654                       ap_scoreboard_image->global->running_generation) { /*
                             restart? */
655                 /* yeah, this could be non-graceful restart, in which case the
656                  * parent will kill us soon enough, but why bother checking?
657                  */
658                 die_now = 1;
659             }
660         apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
661     #else
662
663             /* Check the pod and the generation number after processing a
664              * connection so that we'll go away if a graceful restart occurred
665              * while we were processing the connection or we are the lucky
666              * idle server process that gets to die.
667              */
668             if (ap_mpm_pod_check(pod) == APR_SUCCESS) { /* selected as idle? */
```

```
669             die_now = 1;
670         }
671         else if (ap_my_generation !=
672                  ap_scoreboard_image->global->running_generation) { /*
                        restart? */
673             /* yeah, this could be non-graceful restart, in which case the
674              * parent will kill us soon enough, but why bother checking?
675              */
676             die_now = 1;
677         }
678 #endif
679     }
680     clean_child_exit(0);
681 }

792 #ifdef EL_CONTROL
793 static void perform_idle_server_maintenance(apr_pool_t *p)
794 {
795   int i;
796   int children_to_kill;
797   int children_to_start;
798   int current_slot;
799
800   apr_proc_mutex_lock(ap_scoreboard_image->global->running_info.mutex);
801   if(ap_scoreboard_image->global->running_info.running >
        ap_scoreboard_image->global->want_running) {
802       children_to_kill = ap_scoreboard_image->global->running_info.running -
            ap_scoreboard_image->global->want_running;
803       apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
804       ap_mpm_pod_killpg(pod, children_to_kill);
805     } else if (ap_scoreboard_image->global->running_info.running <
          ap_scoreboard_image->global->want_running) {
806           children_to_start = ap_scoreboard_image->global->want_running -
                ap_scoreboard_image->global->running_info.running;
807         current_slot = ap_scoreboard_image->global->running_info.first_free;
808
809         apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
810         for (i = 0; i < children_to_start; ++i) {
811           make_child(ap_server_conf, current_slot);
812         current_slot =
                ap_scoreboard_image->servers[current_slot][0].next_free;
813       }
814     } else {
815     apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
816   }
817 }
818 #else
819 static void perform_idle_server_maintenance(apr_pool_t *p)
820 {
821     int i;
822     int to_kill;
823     int idle_count;
824     worker_score *ws;
825     int free_length;
826     int free_slots[MAX_SPAWN_RATE];
827     int last_non_dead;
828     int total_non_dead;
829
830     /* initialize the free_list */
```

```
831       free_length = 0;
832
833       to_kill = -1;
834       idle_count = 0;
835       last_non_dead = -1;
836       total_non_dead = 0;
837
838       for (i = 0; i < ap_daemons_limit; ++i) {
839
840     int status;
841
842     if (i >= ap_max_daemons_limit && free_length == idle_spawn_rate)
843       break;
844     ws = &ap_scoreboard_image->servers[i][0];
845     status = ws->status;
846     if (status == SERVER_DEAD) {
847         /* try to keep children numbers as low as possible */
848         if (free_length < idle_spawn_rate) {
849       free_slots[free_length] = i;
850       ++free_length;
851         }
852     }
853     else {
854         /* We consider a starting server as idle because we started it
855          * at least a cycle ago, and if it still hasn't finished starting
856          * then we're just going to swamp things worse by forking more.
857          * So we hopefully won't need to fork more if we count it.
858          * This depends on the ordering of SERVER_READY and SERVER_STARTING.
859          */
860         if (status <= SERVER_READY) {
861       ++ idle_count;
862       /* always kill the highest numbered child if we have to...
863        * no really well thought out reason ... other than observing
864        * the server behaviour under linux where lower numbered children
865        * tend to service more hits (and hence are more likely to have
866        * their data in cpu caches).
867        */
868       to_kill = i;
869         }
870
871         ++total_non_dead;
872         last_non_dead = i;
873     }
874       }
875         ap_max_daemons_limit = last_non_dead + 1;
876
877       if (idle_count > ap_daemons_max_free) {
878     /* kill off one child... we use the pod because that'll cause it to
879      * shut down gracefully, in case it happened to pick up a request
880      * while we were counting
881      */
882     ap_mpm_pod_signal(pod);
883     idle_spawn_rate = 1;
884       }
885       else if (idle_count < ap_daemons_min_free) {
886     /* terminate the free list */
887     if (free_length == 0) {
888         /* only report this condition once */
```

```
889         static int reported = 0;
890
891         if (!reported) {
892         ap_log_error(APLOG_MARK, APLOG_ERR, 0, ap_server_conf,
893                      "server reached MaxClients setting, consider"
894                      " raising the MaxClients setting");
895         reported = 1;
896         }
897         idle_spawn_rate = 1;
898       }
899       else {
900         if (idle_spawn_rate >= 8) {
901         ap_log_error(APLOG_MARK, APLOG_INFO, 0, ap_server_conf,
902                      "server seems busy, (you may need "
903                      "to increase StartServers, or Min/MaxSpareServers), "
904                      "spawning %d children, there are %d idle, and "
905                      "%d total children", idle_spawn_rate,
906                      idle_count, total_non_dead);
907         }
908
909         for (i = 0; i < free_length; ++i) {
910 #ifdef TPF
911           if (make_child(ap_server_conf, free_slots[i]) == -1) {
912               if(free_length == 1) {
913                   shutdown_pending = 1;
914                   ap_log_error(APLOG_MARK, APLOG_EMERG, 0, ap_server_conf,
915                   "No active child processes: shutting down");
916               }
917           }
918 #else
919         make_child(ap_server_conf, free_slots[i]);
920 #endif /* TPF */
921         }
922         /* the next time around we want to spawn twice as many if this
923          * wasn't good enough, but not if we've just done a graceful
924          */
925
926         if (hold_off_on_exponential_spawning) {
927         --hold_off_on_exponential_spawning;
928         }
929         else if (idle_spawn_rate < MAX_SPAWN_RATE) {
930         idle_spawn_rate *= 2;
931         }
932       }
933     }
934     else {
935       idle_spawn_rate = 1;
936     }
937 }
938 #endif

940 /*****************************************************************
941  * Executive routines.
942  */
943
944 int ap_mpm_run(apr_pool_t *_pconf, apr_pool_t *plog, server_rec *s)
945 {
946     int index;
947     int remaining_children_to_start;
```

```
948     apr_status_t rv;
949
950     ap_log_pid(pconf, ap_pid_fname);
951
952     // set the max clients value
953     first_server_limit = server_limit;
954     if (changed_limit_at_restart) {
955         ap_log_error(APLOG_MARK, APLOG_WARNING, 0, s,
956                      "WARNING: Attempt to change ServerLimit "
957                      "ignored during restart");
958         changed_limit_at_restart = 0;
959     }
960
961     /* Initialize cross-process accept lock */
962     ap_lock_fname = apr_psprintf(_pconf, "%s.%" APR_PID_T_FMT,
963                                  ap_server_root_relative(_pconf,
                                         ap_lock_fname),
964                                  ap_my_pid);
965
966     rv = apr_proc_mutex_create(&accept_mutex, ap_lock_fname,
967                                ap_accept_lock_mech, _pconf);
968     if (rv != APR_SUCCESS) {
969         ap_log_error(APLOG_MARK, APLOG_EMERG, rv, s,
970                      "Couldn't create accept lock");
971         mpm_state = AP_MPMQ_STOPPING;
972         return 1;
973     }
974
975 #if APR_USE_SYSVSEM_SERIALIZE
976     if (ap_accept_lock_mech == APR_LOCK_DEFAULT ||
977         ap_accept_lock_mech == APR_LOCK_SYSVSEM) {
978 #else
979     if (ap_accept_lock_mech == APR_LOCK_SYSVSEM) {
980 #endif
981         rv = unixd_set_proc_mutex_perms(accept_mutex);
982         if (rv != APR_SUCCESS) {
983             ap_log_error(APLOG_MARK, APLOG_EMERG, rv, s,
984                          "Couldn't set permissions on cross-process lock; "
985                          "check User and Group directives");
986             mpm_state = AP_MPMQ_STOPPING;
987             return 1;
988         }
989     }
990
991     if (!is_graceful) {
992         if (ap_run_pre_mpm(s->process->pool, SB_SHARED) != OK) {
993             mpm_state = AP_MPMQ_STOPPING;
994             return 1;
995         }
996         /* fix the generation number in the global score; we just got a new,
997          * cleared scoreboard
998          */
999         ap_scoreboard_image->global->running_generation = ap_my_generation;
1000    }
1001
1002    set_signals();
1003
1004    if (one_process) {
```

```
1005            AP_MONCONTROL(1);
1006        }
1007
1008    #ifdef EL_CONTROL
1009        ap_scoreboard_image->global->want_running = ap_daemons_limit;
1010        ap_scoreboard_image->global->keep_alive_timeout =
                ap_server_conf->keep_alive_timeout;
1011
1012        remaining_children_to_start = ap_daemons_limit;
1013        ap_daemons_max_free = ap_daemons_min_free = ap_daemons_limit;
1014
1015    #else
1016        if (ap_daemons_max_free < ap_daemons_min_free + 1)  /* Don't thrash...
                */
1017        ap_daemons_max_free = ap_daemons_min_free + 1;
1018
1019        /* If we're doing a graceful_restart then we're going to see a lot
1020        * of children exiting immediately when we get into the main loop
1021        * below (because we just sent them AP_SIG_GRACEFUL).  This happens pretty
1022        * rapidly... and for each one that exits we'll start a new one until
1023        * we reach at least daemons_min_free.  But we may be permitted to
1024        * start more than that, so we'll just keep track of how many we're
1025        * supposed to start up without the 1 second penalty between each fork.
1026        */
1027        remaining_children_to_start = ap_daemons_to_start;
1028        if (remaining_children_to_start > ap_daemons_limit) {
1029        remaining_children_to_start = ap_daemons_limit;
1030        }
1031    #endif
1032        if (!is_graceful) {
1033        startup_children(remaining_children_to_start);
1034        remaining_children_to_start = 0;
1035        }
1036        else {
1037        /* give the system some time to recover before kicking into
1038            * exponential mode */
1039        hold_off_on_exponential_spawning = 10;
1040        }
1041
1042        ap_log_error(APLOG_MARK, APLOG_NOTICE, 0, ap_server_conf,
1043        "%s configured -- resuming normal operations",
1044        ap_get_server_version());
1045        ap_log_error(APLOG_MARK, APLOG_INFO, 0, ap_server_conf,
1046        "Server built: %s", ap_get_server_built());
1047    #ifdef AP_MPM_WANT_SET_ACCEPT_LOCK_MECH
1048        ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf,
1049        "AcceptMutex: %s (default: %s)",
1050        apr_proc_mutex_name(accept_mutex),
1051        apr_proc_mutex_defname());
1052    #endif
1053        restart_pending = shutdown_pending = 0;
1054
1055        mpm_state = AP_MPMQ_RUNNING;
1056
1057        while (!restart_pending && !shutdown_pending) {
1058        int child_slot;
1059            apr_exit_why_e exitwhy;
1060        int status, processed_status;
```

```
1061            /* this is a memory leak, but I'll fix it later. */
1062        apr_proc_t pid;
1063
1064            ap_wait_or_timeout(&exitwhy, &status, &pid, pconf);
1065
1066        /* XXX: if it takes longer than 1 second for all our children
1067         * to start up and get into IDLE state then we may spawn an
1068         * extra child
1069         */
1070        if (pid.pid != -1) {
1071                processed_status = ap_process_child_status(&pid, exitwhy,
                            status);
1072                if (processed_status == APEXIT_CHILDFATAL) {
1073                    mpm_state = AP_MPMQ_STOPPING;
1074                    return 1;
1075                }
1076
1077            /* non-fatal death... note that it's gone in the scoreboard. */
1078            child_slot = find_child_by_pid(&pid);
1079            if (child_slot >= 0) {
1080          (void) ap_update_child_status_from_indexes(child_slot, 0, SERVER_DEAD,
1081                                                     (request_rec *)
                                                           NULL);
1082                    if (processed_status == APEXIT_CHILDSICK) {
1083                        /* child detected a resource shortage (E[NM]FILE,
                               ENOBUFS, etc)
1084                         * cut the fork rate to the minimum
1085                         */
1086                        idle_spawn_rate = 1;
1087                    }
1088                    else if (remaining_children_to_start
1089              && child_slot < ap_daemons_limit) {
1090              /* we're still doing a 1-for-1 replacement of dead
1091             * children with new children
1092             */
1093                make_child(ap_server_conf, child_slot);
1094                --remaining_children_to_start;
1095            }
1096    #if APR_HAS_OTHER_CHILD
1097            }
1098            else if (apr_proc_other_child_read(&pid, status) == 0) {
1099        /* handled */
1100    #endif
1101            }
1102            else if (is_graceful) {
1103        /* Great, we've probably just lost a slot in the
1104             * scoreboard.  Somehow we don't know about this
1105             * child.
1106             */
1107        ap_log_error(APLOG_MARK, APLOG_WARNING,
1108                                 0, ap_server_conf,
1109                "long lost child came home! (pid %ld)", (long)pid.pid);
1110            }
1111            /* Don't perform idle maintenance when a child dies,
1112         * only do it when there's a timeout.  Remember only a
1113         * finite number of children can die, and it's pretty
1114         * pathological for a lot to die suddenly.
1115         */
```

```
1116            continue;
1117        }
1118        else if (remaining_children_to_start) {
1119            /* we hit a 1 second timeout in which none of the previous
1120             * generation of children needed to be reaped... so assume
1121             * they're all done, and pick up the slack if any is left.
1122             */
1123            startup_children(remaining_children_to_start);
1124            remaining_children_to_start = 0;
1125            /* In any event we really shouldn't do the code below because
1126             * few of the servers we just started are in the IDLE state
1127             * yet, so we'd mistakenly create an extra server.
1128             */
1129            continue;
1130        }
1131
1132        perform_idle_server_maintenance(pconf);
1133    #ifdef TPF
1134            shutdown_pending = os_check_server(tpf_server_name);
1135            ap_check_signals();
1136            sleep(1);
1137    #endif /*TPF */
1138            }
1139
1140            mpm_state = AP_MPMQ_STOPPING;
1141
1142            if (shutdown_pending) {
1143
1144        /* Time to gracefully shut down:
1145         * Kill child processes, tell them to call child_exit, etc...
1146         */
1147        if (unixd_killpg(getpgrp(), SIGTERM) < 0) {
1148            ap_log_error(APLOG_MARK, APLOG_WARNING, errno, ap_server_conf, "killpg
                    SIGTERM");
1149        }
1150
1151        ap_reclaim_child_processes(1);      /* Start with SIGTERM */
1152
1153        /* cleanup pid file on normal shutdown */
1154        {
1155            const char *pidfile = NULL;
1156            pidfile = ap_server_root_relative (pconf, ap_pid_fname);
1157            if ( pidfile != NULL && unlink(pidfile) == 0)
1158          ap_log_error(APLOG_MARK, APLOG_INFO,
1159                0, ap_server_conf,
1160                "removed PID file %s (pid=%ld)",
1161                pidfile, (long)getpid());
1162        }
1163
1164        ap_log_error(APLOG_MARK, APLOG_NOTICE, 0, ap_server_conf,
1165              "caught SIGTERM, shutting down");
1166        return 1;
1167            }
1168
1169            /* we've been told to restart */
1170            apr_signal(SIGHUP, SIG_IGN);
1171            if (one_process) {
1172        /* not worth thinking about */
```

```
1173    return 1;
1174      }
1175
1176      /* advance to the next generation */
1177      /* XXX: we really need to make sure this new generation number isn't in
1178       * use by any of the children.
1179       */
1180      ++ap_my_generation;
1181      ap_scoreboard_image->global->running_generation = ap_my_generation;
1182
1183      if (is_graceful) {
1184    ap_log_error(APLOG_MARK, APLOG_NOTICE, 0, ap_server_conf,
1185          "Graceful restart requested, doing restart");
1186
1187    /* kill off the idle ones */
1188          ap_mpm_pod_killpg(pod, ap_max_daemons_limit);
1189
1190    /* This is mostly for debugging... so that we know what is still
1191        * gracefully dealing with existing request.  This will break
1192        * in a very nasty way if we ever have the scoreboard totally
1193        * file-based (no shared memory)
1194        */
1195    for (index = 0; index < ap_daemons_limit; ++index) {
1196        if (ap_scoreboard_image->servers[index][0].status != SERVER_DEAD) {
1197      ap_scoreboard_image->servers[index][0].status = SERVER_GRACEFUL;
1198        }
1199    }
1200      }
1201      else {
1202    /* Kill 'em off */
1203    if (unixd_killpg(getpgrp(), SIGHUP) < 0) {
1204        ap_log_error(APLOG_MARK, APLOG_WARNING, errno, ap_server_conf, "killpg
              SIGHUP");
1205    }
1206    ap_reclaim_child_processes(0);      /* Not when just starting up */
1207    ap_log_error(APLOG_MARK, APLOG_NOTICE, 0, ap_server_conf,
1208          "SIGHUP received.  Attempting to restart");
1209      }
1210
1211      return 0;
1212  }
```

B.2 scoreboard.h

```
1  /* Copyright 2001-2005 The Apache Software Foundation or its licensors, as
2   * applicable.
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *      http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  * See the License for the specific language governing permissions and
14  * limitations under the License.
```

```
15   */
16
17  #ifndef APACHE_SCOREBOARD_H
18  #define APACHE_SCOREBOARD_H
19
20  #ifdef __cplusplus
21  extern "C" {
22  #endif
23
24  #ifdef HAVE_SYS_TIMES_H
25  #include <sys/time.h>
26  #include <sys/times.h>
27  #elif defined(TPF)
28  #include <time.h>
29  #endif
30
31  #include "ap_config.h"
32  #include "apr_hooks.h"
33  #include "apr_thread_proc.h"
34  #include "apr_portable.h"
35  #include "apr_shm.h"
36
37  /* Scoreboard file, if there is one */
38  #ifndef DEFAULT_SCOREBOARD
39  #define DEFAULT_SCOREBOARD "logs/apache_runtime_status"
40  #endif
41
42  //Activate Control
43  #ifndef EL_CONTROL
44  #define EL_CONTROL 1
45  #endif
46  #include <sys/socket.h>
47  #include <sys/types.h>
48  #include <netinet/tcp.h>
49  #include "../srclib/apr/include/arch/unix/apr_arch_networkio.h"
50
51  /* Scoreboard info on a process is, for now, kept very brief ---
52   * just status value and pid (the latter so that the caretaker process
53   * can properly update the scoreboard when a process dies).  We may want
54   * to eventually add a separate set of long_score structures which would
55   * give, for each process, the number of requests serviced, and info on
56   * the current, or most recent, request.
57   *
58   * Status values:
59   */
60
61  #define SERVER_DEAD 0
62  #define SERVER_STARTING 1 /* Server Starting up */
63  #define SERVER_READY 2    /* Waiting for connection (or accept() lock) */
64  #define SERVER_BUSY_READ 3  /* Reading a client request */
65  #define SERVER_BUSY_WRITE 4 /* Processing a client request */
66  #define SERVER_BUSY_KEEPALIVE 5 /* Waiting for more requests via keepalive
        */
67  #define SERVER_BUSY_LOG 6 /* Logging the request */
68  #define SERVER_BUSY_DNS 7 /* Looking up a hostname */
69  #define SERVER_CLOSING 8  /* Closing the connection */
70  #define SERVER_GRACEFUL 9 /* server is gracefully finishing request */
71  #define SERVER_IDLE_KILL 10     /* Server is cleaning up idle children. */
```

```
72 #define SERVER_NUM_STATUS 11  /* number of status settings */
73
74 /* Type used for generation indicies.  Startup and every restart cause a
75  * new generation of children to be spawned.  Children within the same
76  * generation share the same configuration information -- pointers to stuff
77  * created at config time in the parent are valid across children.  However,
78  * this can't work effectively with non-forked architectures.  So while the
79  * arrays in the scoreboard never change between the parent and forked
80  * children, so they do not require shm storage, the contents of the shm
81  * may contain no pointers.
82  */
83 typedef int ap_generation_t;
84
85 /* Is the scoreboard shared between processes or not?
86  * Set by the MPM when the scoreboard is created.
87  */
88 typedef enum {
89     SB_NOT_SHARED = 1,
90     SB_SHARED = 2
91 } ap_scoreboard_e;
92
93 #define SB_WORKING  0  /* The server is busy and the child is useful. */
94 #define SB_IDLE_DIE 1  /* The server is idle and the child is superfluous.
    */
95                        /*   The child should check for this and exit
                               gracefully. */
96
97 /* stuff which is worker specific */
98 /*********************WARNING*************************************/
99 /* These are things that are used by mod_status. Do not put anything */
100 /*   in here that you cannot live without. This structure will not   */
101 /*   be available if mod_status is not loaded.                       */
102 /*****************************************************************/
103 typedef struct worker_score worker_score;
104
105 struct worker_score {
106     int thread_num;
107 #if APR_HAS_THREADS
108     apr_os_thread_t tid;
109 #endif
110     unsigned char status;
111     unsigned long access_count;
112     apr_off_t     bytes_served;
113     unsigned long my_access_count;
114     apr_off_t     my_bytes_served;
115     apr_off_t     conn_bytes;
116     unsigned short conn_count;
117     apr_time_t start_time;
118     apr_time_t stop_time;
119     apr_socket_t *sb_socket;
120 #ifdef HAVE_TIMES
121     struct tms times;
122 #endif
123     apr_time_t last_used;
124     char client[32];    /* Keep 'em small... */
125     char request[64];   /* We just want an idea... */
126     char vhost[32];         /* What virtual host is being accessed? */
127 #ifdef EL_CONTROL
```

```
128   int next_free;
129   int keepalives;
130   int rounds;
131 #endif
132 };
133
134 #ifdef EL_CONTROL
135
136 typedef struct {
137     int          nbr_of_requests;  // total nbr of requests during one sample
          period
138     apr_time_t       requests_time;    // total time it took to process all
          these requests
139     int          rtt;         // total estimated round-trip time
140     int          rtt_var;       // total mean deviation maximum for the rtt;
141     int          min_keepalives;    // min nbr of keep-alive rounds
142     int          max_keepalives;    // max nbr of keep-alive rounds
143     int          keepalives;      // total nbr of keep-alive rounds
144     apr_proc_mutex_t  *mutex;        // must be owned to access the above
          fields
145 } request_info;
146
147 typedef struct {
148     int          running;  // nbr of child-processess
149     int          first_free; // index of the first free scoreboard slot
150     apr_proc_mutex_t  *mutex;   // must be owned to access the above fields
151 } running_info;
152
153 typedef struct {
154     int          running_idle; // nbr of child-processess that are idle
155     apr_proc_mutex_t  *mutex;      // must be owned to access the above
          fields
156 } running_idle_info;
157
158 struct control_logdata {
159     int          nbr_of_requests;  // total nbr of requests during one sample
          period
160     apr_time_t       requests_time;    // total time it took to process all
          the requests
161     int          rtt;
162     int          rtt_var;
163     int          running;       // nbr of child-processess running
164     int          running_idle;      // nbr of child-processess that are idle
165     int          min_keepalives;    // min nbr of keep-alive rounds
166     int          max_keepalives;    // max nbr of keep-alive rounds
167     int          keepalives;      // total nbr of keep-alive rounds
168     int          min_rounds;      // min nbr of child rounds
169     int          max_rounds;      // max nbr of child rounds
170     int          rounds;        // total nbr of child rounds
171 };
172
173 typedef struct {
174     int          min_rounds;    // min nbr of child rounds
175     int          max_rounds;    // max nbr of child rounds
176     int          rounds;      // total nbr of child rounds
177     apr_proc_mutex_t  *mutex;      // must be owned to access the above
          fields
178 } child_info;
```

```
179
180 typedef struct {
181     int          want_running; // nbr of child-processess that we want
182     int          test_number;  // The number of the test
183     apr_proc_mutex_t  *mutex;      // must be owned to access the above
            fields
184 } logger_test;
185 #endif
186
187 typedef struct {
188     int             server_limit;
189     int             thread_limit;
190     ap_scoreboard_e sb_type;
191     ap_generation_t running_generation; /* the generation of children which
192                                          * should still be serving requests.
                                                */
193     apr_time_t restart_time;
194 #ifdef ELCONTROL
195   running_info running_info;
196   running_idle_info running_idle_info;
197   int want_running;
198   apr_interval_time_t keep_alive_timeout;
199   request_info request_info;
200   child_info child_info;
201   logger_test logger_test;
202   int max_keepalives_child;
203   int min_keepalives_child;
204   int max_rounds_child;
205   int min_rounds_child;
206   //int new_test;
207   //int stop_test;
208   int test_stop_code;   // 1 = stop single test, 2 = stop logger
209   //int new_mcvalue;
210   //int new_testvalue;
211 #endif
212 } global_score;
213
214 /* stuff which the parent generally writes and the children rarely read */
215 typedef struct process_score process_score;
216 struct process_score{
217     pid_t pid;
218     ap_generation_t generation; /* generation of this child */
219     ap_scoreboard_e sb_type;
220     int quiescing;          /* the process whose pid is stored above is
221                              * going down gracefully
222                              */
223 };
224
225 /* Scoreboard is now in 'local' memory, since it isn't updated once created,
226  * even in forked architectures.  Child created-processes (non-fork) will
227  * set up these indicies into the (possibly relocated) shmem records.
228  */
229 typedef struct {
230     global_score *global;
231     process_score *parent;
232     worker_score **servers;
233 } scoreboard;
234
```

```
235 typedef struct ap_sb_handle_t ap_sb_handle_t;
236
237 AP_DECLARE(int) ap_exists_scoreboard_image(void);
238 AP_DECLARE(void) ap_increment_counts(ap_sb_handle_t *sbh, request_rec *r);
239
240 int ap_create_scoreboard(apr_pool_t *p, ap_scoreboard_e t);
241 apr_status_t ap_reopen_scoreboard(apr_pool_t *p, apr_shm_t **shm, int
        detached);
242 void ap_init_scoreboard(void *shared_score);
243 AP_DECLARE(int) ap_calc_scoreboard_size(void);
244 apr_status_t ap_cleanup_scoreboard(void *d);
245
246 AP_DECLARE(void) ap_create_sb_handle(ap_sb_handle_t **new_sbh, apr_pool_t
        *p,
247                                      int child_num, int thread_num);
248
249 AP_DECLARE(int) find_child_by_pid(apr_proc_t *pid);
250 AP_DECLARE(int) ap_update_child_status(ap_sb_handle_t *sbh, int status,
        request_rec *r);
251 AP_DECLARE(int) ap_update_child_status_from_indexes(int child_num, int
        thread_num,
252                                                     int status, request_rec
                                                        *r);
253
254 #ifdef EL_CONTROL
255 void ap_time_process_request(request_rec *r, int status);
256 AP_DECLARE(int) ap_init_control_data(void);
257 AP_DECLARE(void) ap_get_control_logdata(struct control_logdata *data);
258 AP_DECLARE(void) ap_set_backlog(int value);
259 AP_DECLARE(void) ap_update_child_rounds(ap_sb_handle_t *sbh);
260 AP_DECLARE(void) ap_set_want_running(int value);
261 AP_DECLARE(void) ap_set_keep_alive_timeout(int value);
262 #else
263 void ap_time_process_request(ap_sb_handle_t *sbh, int status);
264 #endif
265
266 AP_DECLARE(worker_score *) ap_get_scoreboard_worker(int x, int y);
267 AP_DECLARE(process_score *) ap_get_scoreboard_process(int x);
268 AP_DECLARE(global_score *) ap_get_scoreboard_global(void);
269
270 AP_DECLARE_DATA extern scoreboard *ap_scoreboard_image;
271 AP_DECLARE_DATA extern const char *ap_scoreboard_fname;
272 AP_DECLARE_DATA extern int ap_extended_status;
273
274 AP_DECLARE_DATA extern ap_generation_t volatile ap_my_generation;
275
276 /* Hooks */
277 /**
278   * Hook for post scoreboard creation, pre mpm.
279   * @param p        Apache pool to allocate from.
280   * @param sb_type
281   * @ingroup hooks
282   * @return OK or DECLINE on success; anything else is a error
283   */
284 AP_DECLARE_HOOK(int, pre_mpm, (apr_pool_t *p, ap_scoreboard_e sb_type))
285
286 /* for time_process_request() in http_main.c */
287 #define START_PREQUEST 1
```

```
288 #define STOP_PREQUEST   2
289
290 #ifdef __cplusplus
291 }
292 #endif
293
294 #endif  /* !APACHE_SCOREBOARD_H */
```

B.3 scoreboard.c

```
101 void ap_init_scoreboard(void *shared_score)
102 {
103     char *more_storage;
104     int i;
105
106     ap_calc_scoreboard_size();
107     ap_scoreboard_image =
108         calloc(1, sizeof(scoreboard) + server_limit * sizeof(worker_score
                *));
109     more_storage = shared_score;
110     ap_scoreboard_image->global = (global_score *)more_storage;
111     more_storage += sizeof(global_score);
112     ap_scoreboard_image->parent = (process_score *)more_storage;
113     more_storage += sizeof(process_score) * server_limit;
114     ap_scoreboard_image->servers =
115         (worker_score **)((char*)ap_scoreboard_image + sizeof(scoreboard));
116     for (i = 0; i < server_limit; i++) {
117         ap_scoreboard_image->servers[i] = (worker_score *)more_storage;
118 #ifdef EL_CONTROL
119         ap_scoreboard_image->servers[i][0].next_free = (i + 1);
120 #endif
121         more_storage += thread_limit * sizeof(worker_score);
122     }
123
124     ap_assert(more_storage == (char*)shared_score + scoreboard_size);
125     ap_scoreboard_image->global->server_limit = server_limit;
126     ap_scoreboard_image->global->thread_limit = thread_limit;
127 }

252 /* Create or reinit an existing scoreboard. The MPM can control whether
253  * the scoreboard is shared across multiple processes or not
254  */
255 int ap_create_scoreboard(apr_pool_t *p, ap_scoreboard_e sb_type)
256 {
257     int running_gen = 0;
258     int i;
259 #if APR_HAS_SHARED_MEMORY
260     apr_status_t rv;
261 #endif
262
263     if (ap_scoreboard_image) {
264         running_gen = ap_scoreboard_image->global->running_generation;
265         ap_scoreboard_image->global->restart_time = apr_time_now();
266         memset(ap_scoreboard_image->parent, 0,
267                sizeof(process_score) * server_limit);
268         for (i = 0; i < server_limit; i++) {
269             memset(ap_scoreboard_image->servers[i], 0,
270                    sizeof(worker_score) * thread_limit);
```

```
271             }
272             return OK;
273         }
274 
275         ap_calc_scoreboard_size();
276 #if APR_HAS_SHARED_MEMORY
277         if (sb_type == SB_SHARED) {
278             void *sb_shared;
279             rv = open_scoreboard(p);
280             if (rv || !(sb_shared = apr_shm_baseaddr_get(ap_scoreboard_shm))) {
281                 return HTTP_INTERNAL_SERVER_ERROR;
282             }
283             memset(sb_shared, 0, scoreboard_size);
284             ap_init_scoreboard(sb_shared);
285         }
286         else
287 #endif
288         {
289             /* A simple malloc will suffice */
290             void *sb_mem = calloc(1, scoreboard_size);
291             if (sb_mem == NULL) {
292                 ap_log_error(APLOG_MARK, APLOG_CRIT, 0, NULL,
293                              "(%d)%s: cannot allocate scoreboard",
294                              errno, strerror(errno));
295                 return HTTP_INTERNAL_SERVER_ERROR;
296             }
297             ap_init_scoreboard(sb_mem);
298         }
299 
300 #ifdef ELCONTROL
301       //The last one shouldn't link to anyone
302         ap_scoreboard_image->servers[(server_limit - 1)][0].next_free = -1;
303 
304       //Initialize the running_info struct
305       ap_scoreboard_image->global->running_info.first_free = 0;
306       ap_scoreboard_image->global->running_info.running = 0;
307       apr_proc_mutex_create(&(ap_scoreboard_image->global->running_info).mutex,
            "running_info", APR_LOCK_DEFAULT, p);
308 
309       //Initialize the running_idle_info struct
310       ap_scoreboard_image->global->running_idle_info.running_idle = 0;
311       apr_proc_mutex_create(&(ap_scoreboard_image->global->running_idle_info).mutex,
             "running_idle_info", APR_LOCK_DEFAULT, p);
312 
313       //Initialize the logger_test struct mutex
314       apr_proc_mutex_create(&(ap_scoreboard_image->global->logger_test).mutex,
            "logger_test", APR_LOCK_DEFAULT, p);
315 
316         // Make logger wait for new test by locking the mutex
317         apr_proc_mutex_lock(ap_scoreboard_image->global->logger_test.mutex);
318 
319         //Initialize the request_idle_info struct mutex
320       apr_proc_mutex_create(&(ap_scoreboard_image->global->request_info).mutex,
            "request_info", APR_LOCK_DEFAULT, p);
321 
322       // Initialize the child_info struct mutex
323       apr_proc_mutex_create(&(ap_scoreboard_image->global->child_info).mutex,
            "child_info", APR_LOCK_DEFAULT, p);
```

```
324
325     ap_init_control_data();
326
327     //Start logger process
328       logger_init();
329
330       //Start controller process
331       //controller_init();
332   #endif
333       ap_scoreboard_image->global->sb_type = sb_type;
334       ap_scoreboard_image->global->running_generation = running_gen;
335       ap_scoreboard_image->global->restart_time = apr_time_now();
336
337       apr_pool_cleanup_register(p, NULL, ap_cleanup_scoreboard,
              apr_pool_cleanup_null);
338
339       return OK;
340   }

399   AP_DECLARE(int) ap_update_child_status_from_indexes(int child_num,
400                                                       int thread_num,
401                                                       int status,
402                                                       request_rec *r)
403   {
404       int old_status;
405       worker_score *ws;
406       process_score *ps;
407
408       if (child_num < 0) {
409           return -1;
410       }
411
412       ws = &ap_scoreboard_image->servers[child_num][thread_num];
413       old_status = ws->status;
414       ws->status = status;
415
416       ps = &ap_scoreboard_image->parent[child_num];
417
418       if (status == SERVER_READY
419           && old_status == SERVER_STARTING) {
420           ws->thread_num = child_num * thread_limit + thread_num;
421           ps->generation = ap_my_generation;
422       }
423
424   #ifdef ELCONTROL
425     if(status == SERVER_STARTING) {
426       // Update the running_info struct
427       apr_proc_mutex_lock(ap_scoreboard_image->global->running_info.mutex);
428         ap_scoreboard_image->global->running_info.running++;
429         ap_scoreboard_image->global->running_info.first_free = ws->next_free;
430       apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
431
432       // Update the running_idle_info struct
433       apr_proc_mutex_lock(ap_scoreboard_image->global->running_idle_info.mutex);
434         ap_scoreboard_image->global->running_idle_info.running_idle++;
435       apr_proc_mutex_unlock(ap_scoreboard_image->global->running_idle_info.mutex);
436     } else if(status == SERVER_DEAD) {
437       // Update the running_info struct
438       apr_proc_mutex_lock(ap_scoreboard_image->global->running_info.mutex);
```

```
439         ap_scoreboard_image->global->running_info.running--;
440         int first_free = ap_scoreboard_image->global->running_info.first_free;
441         ws->next_free = first_free;
442         ap_scoreboard_image->global->running_info.first_free = child_num;
443       apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
444
445       // Update the request_info struct
446       apr_proc_mutex_lock(ap_scoreboard_image->global->request_info.mutex);
447         ap_scoreboard_image->global->request_info.keepalives -=
              ws->keepalives;
448         ws->keepalives = 0;
449         if(ap_scoreboard_image->global->min_keepalives_child == child_num) {
450           ap_scoreboard_image->global->request_info.min_keepalives = 0;
451           ap_scoreboard_image->global->min_keepalives_child = -1;
452         }
453
454         if(ap_scoreboard_image->global->max_keepalives_child == child_num) {
455           ap_scoreboard_image->global->request_info.max_keepalives = 0;
456           ap_scoreboard_image->global->max_keepalives_child = -1;
457         }
458       apr_proc_mutex_unlock(ap_scoreboard_image->global->request_info.mutex);
459
460       // Update the child_info struct
461       apr_proc_mutex_lock(ap_scoreboard_image->global->child_info.mutex);
462         ap_scoreboard_image->global->child_info.rounds -= ws->rounds;
463         ws->rounds = 0;
464         if(ap_scoreboard_image->global->min_rounds_child == child_num) {
465           ap_scoreboard_image->global->child_info.min_rounds = 0;
466           ap_scoreboard_image->global->min_rounds_child = -1;
467         }
468
469         if(ap_scoreboard_image->global->max_rounds_child == child_num) {
470           ap_scoreboard_image->global->child_info.max_rounds = 0;
471           ap_scoreboard_image->global->max_rounds_child = -1;
472         }
473       apr_proc_mutex_unlock(ap_scoreboard_image->global->child_info.mutex);
474
475     } else if(old_status > SERVER_READY && status == SERVER_READY) {
476       // Update the running_idle_info struct
477       apr_proc_mutex_lock(ap_scoreboard_image->global->running_idle_info.mutex);
478         ap_scoreboard_image->global->running_idle_info.running_idle++;
479       apr_proc_mutex_unlock(ap_scoreboard_image->global->running_idle_info.mutex);
480     } else if(old_status == SERVER_READY && status > SERVER_READY) {
481       // Update the running_idle_info struct
482       apr_proc_mutex_lock(ap_scoreboard_image->global->running_idle_info.mutex);
483         ap_scoreboard_image->global->running_idle_info.running_idle--;
484       apr_proc_mutex_unlock(ap_scoreboard_image->global->running_idle_info.mutex);
485     } else if(status == SERVER_BUSY_WRITE) {
486       if(r) {
487         if(r->args) {
488         if(strncmp(r->args ,"wr=", 3) == 0) {
489           //fprintf(stderr, "WR discovered: %s status: %i\n", r->args,
                status);
490           //fflush(stderr);
491           char running_value[5];
492           char test_number_value[5];
493           int running_finished = 0, wri = 0, testi = 0, argsi;
494           for(argsi = 3; r->args[argsi] != '\0'; argsi++) {
```

```
495             if(r->args[argsi] != '&' && !running_finished) {
496               running_value[wri++] = r->args[argsi];
497             } else if (r->args[argsi] == '&') {
498               argsi += 5;
499               running_finished = 1;
500             } else {
501               test_number_value[testi++] = r->args[argsi];
502             }
503           }
504           ap_scoreboard_image->global->logger_test.want_running =
                  atoi(running_value);
505           ap_scoreboard_image->global->logger_test.test_number =
                  atoi(test_number_value);
506
507           ap_init_control_data();
508
509           // Start logger by releasing the mutex
510           apr_proc_mutex_unlock(ap_scoreboard_image->global->logger_test.mutex);
511         } else if(strcmp(r->args ,"stoptest") == 0) {
512           //fprintf(stderr, "STOP TEST discoverd at status: %i!!\n", status);
513           //fflush(stderr);
514           // Stop logger by locking the mutex and setting a stop code
515           if(ap_scoreboard_image->global->test_stop_code == 0) {
516             apr_proc_mutex_lock(ap_scoreboard_image->global->logger_test.mutex);
517           }
518             ap_scoreboard_image->global->test_stop_code = 1;
519         } else if(strcmp(r->args ,"stopwholetest") == 0) {
520           // Stop logger by locking the mutex and setting a stop code
521           if(ap_scoreboard_image->global->test_stop_code == 0) {
522             apr_proc_mutex_lock(ap_scoreboard_image->global->logger_test.mutex);
523           }
524             ap_scoreboard_image->global->test_stop_code = 2;
525           // Make logger run again by releasing the mutex, the logger will
526           // immediately examine the stop code and as it is 2, it will
                  terminate
527           apr_proc_mutex_unlock(ap_scoreboard_image->global->logger_test.mutex);
528         } else if(strncmp(r->args ,"backlog=", 8) == 0) {
529           char backlog_value[5];
530           int argsi, bi = 0;
531           for(argsi = 8; r->args[argsi] != '\0'; argsi++) {
532             backlog_value[bi++] = r->args[argsi];
533           }
534           //fprintf(stderr, "BACKLOG discoverd %i!!\n", atoi(backlog_value));
535           //fflush(stderr);
536           ap_set_backlog(atoi(backlog_value));
537         }
538       }}
539     }
540 #endif
541
542       if (ap_extended_status) {
543           ws->last_used = apr_time_now();
544           if (status == SERVER_READY || status == SERVER_DEAD) {
545               /*
546                * Reset individual counters
547                */
548               if (status == SERVER_DEAD) {
549                   ws->my_access_count = 0L;
```

```
550                 ws->my_bytes_served = 0L;
551             }
552             ws->conn_count = 0;
553             ws->conn_bytes = 0;
554         }
555         if (r) {
556             conn_rec *c = r->connection;
557             apr_cpystrn(ws->client, ap_get_remote_host(c, r->per_dir_config,
558                         REMOTE_NOLOOKUP, NULL), sizeof(ws->client));
559             if (r->the_request == NULL) {
560                 apr_cpystrn(ws->request, "NULL", sizeof(ws->request));
561             } else if (r->parsed_uri.password == NULL) {
562                 apr_cpystrn(ws->request, r->the_request,
                        sizeof(ws->request));
563             } else {
564                 /* Don't reveal the password in the server-status view */
565                 apr_cpystrn(ws->request, apr_pstrcat(r->pool, r->method, "
                        ",
566                                 apr_uri_unparse(r->pool, &r->parsed_uri,
567                                 APR_URI_UNP_OMITPASSWORD),
568                                 r->assbackwards ? NULL : " ", r->protocol,
                                    NULL),
569                                 sizeof(ws->request));
570             }
571             apr_cpystrn(ws->vhost, r->server->server_hostname,
572                         sizeof(ws->vhost));
573         }
574     }
575
576     return old_status;
577 }

585 #ifdef EL_CONTROL
586 void ap_time_process_request(request_rec *r, int status)
587 {
588 ap_sb_handle_t *sbh = r->connection->sbh;
589 #else
590 void ap_time_process_request(ap_sb_handle_t *sbh, int status)
591 {
592 #endif
593     worker_score *ws;
594
595     if (sbh->child_num < 0) {
596         return;
597     }
598
599     ws = &ap_scoreboard_image->servers[sbh->child_num][sbh->thread_num];
600
601     if (status == START_PREQUEST) {
602         ws->start_time = apr_time_now();
603 #ifdef EL_CONTROL
604   struct tcp_info tcpinfo;
605   socklen_t socklen = sizeof(tcpinfo);
606   getsockopt(ws->sb_socket->socketdes, SOL_TCP, TCP_INFO, &tcpinfo,
          &socklen);
607 #endif
608     }
609     else if (status == STOP_PREQUEST) {
610         ws->stop_time = apr_time_now();
```

```
611 #ifdef ELCONTROL
612   struct tcp_info tcpinfo;
613   socklen_t socklen = sizeof(tcpinfo);
614   getsockopt(ws->sb_socket->socketdes, SOL_TCP, TCP_INFO, &tcpinfo,
          &socklen);
615
616   apr_proc_mutex_lock(ap_scoreboard_image->global->request_info.mutex);
617     ap_scoreboard_image->global->request_info.nbr_of_requests++;
618     ap_scoreboard_image->global->request_info.requests_time +=
            (ws->stop_time - ws->start_time);
619     ap_scoreboard_image->global->request_info.rtt += tcpinfo.tcpi_rtt;
620     ap_scoreboard_image->global->request_info.rtt_var +=
            tcpinfo.tcpi_rttvar;
621
622     if(ap_scoreboard_image->global->request_info.keepalives == 0) {
623       ap_scoreboard_image->global->request_info.keepalives =
              r->connection->keepalives;
624     } else {
625       ap_scoreboard_image->global->request_info.keepalives +=
              r->connection->keepalives - ws->keepalives;
626       ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Keepalives
              updated to: %i change: %i",
              ap_scoreboard_image->global->request_info.keepalives,
              (r->connection->keepalives - ws->keepalives));
627     }
628
629     if(r->connection->keepalives <=
          ap_scoreboard_image->global->request_info.min_keepalives ||
          ap_scoreboard_image->global->min_keepalives_child == sbh->child_num
          || ap_scoreboard_image->global->min_keepalives_child == -1) {
630       ap_scoreboard_image->global->request_info.min_keepalives =
              r->connection->keepalives;
631       ap_scoreboard_image->global->min_keepalives_child = sbh->child_num;
632       ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Min
              keepalives updated to: %i by child: %i",
              ap_scoreboard_image->global->request_info.min_keepalives,
              ap_scoreboard_image->global->min_keepalives_child);
633     }
634
635     if(r->connection->keepalives >=
          ap_scoreboard_image->global->request_info.max_keepalives ||
          ap_scoreboard_image->global->max_keepalives_child == sbh->child_num
          || ap_scoreboard_image->global->max_keepalives_child == -1) {
636       ap_scoreboard_image->global->request_info.max_keepalives =
              r->connection->keepalives;
637       ap_scoreboard_image->global->max_keepalives_child = sbh->child_num;
638       ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Max
              keepalives updated to: %i by child: %i",
              ap_scoreboard_image->global->request_info.max_keepalives,
              ap_scoreboard_image->global->max_keepalives_child);
639     }
640
641     ws->keepalives = r->connection->keepalives;
642     //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Score
          Response time: %qd total: %qd", (ws->stop_time -
          ws->start_time),ap_scoreboard_image->global->requesttime_info.requests_time);
643     //ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Score
          Response time: %qd", (ws->stop_time - ws->start_time));
```

```
644     apr_proc_mutex_unlock(ap_scoreboard_image->global->request_info.mutex);
645 #endif
646         }
647         //fprintf(stderr,"\nSET TIME: %d!!!\n", apr_time_now());
648 }

672 #ifdef ELCONTROL
673 AP_DECLARE(int) ap_init_control_data(void)
674 {
675     //Initialize the request_idle_info struct
676     apr_proc_mutex_lock(ap_scoreboard_image->global->request_info.mutex);
677       ap_scoreboard_image->global->request_info.nbr_of_requests = 0;
678       ap_scoreboard_image->global->request_info.requests_time = 0;
679       ap_scoreboard_image->global->request_info.rtt = 0;
680       ap_scoreboard_image->global->request_info.rtt_var = 0;
681       ap_scoreboard_image->global->request_info.keepalives = 0;
682       ap_scoreboard_image->global->request_info.min_keepalives = 0;
683       ap_scoreboard_image->global->request_info.max_keepalives = 0;
684     apr_proc_mutex_unlock(ap_scoreboard_image->global->request_info.mutex);
685
686     // Initialize the child_info struct
687     apr_proc_mutex_lock(ap_scoreboard_image->global->child_info.mutex);
688       ap_scoreboard_image->global->child_info.min_rounds = 0;
689       ap_scoreboard_image->global->child_info.max_rounds = 0;
690       ap_scoreboard_image->global->child_info.rounds = 0;
691     apr_proc_mutex_unlock(ap_scoreboard_image->global->child_info.mutex);
692
693     // Initialize the global parameters
694     //ap_scoreboard_image->global->keep_alive_timeout = -1;
695       ap_scoreboard_image->global->min_keepalives_child = -1;
696       ap_scoreboard_image->global->max_keepalives_child = -1;
697       ap_scoreboard_image->global->min_rounds_child = -1;
698       ap_scoreboard_image->global->max_rounds_child = -1;
699       ap_scoreboard_image->global->test_stop_code = 0;
700
701       int i;
702       for ( i= 0; i < server_limit; i++) {
703           ap_scoreboard_image->servers[i][0].keepalives = 0;
704           ap_scoreboard_image->servers[i][0].rounds = 0;
705       }
706 }
707
708 AP_DECLARE(void) ap_set_want_running(int value)
709 {
710     ap_scoreboard_image->global->want_running = value;
711
712     //perform_idle_server_maintenance(pconf);
713       /*
714     int old_limit, diff;
715     old_limit =  ap_daemons_limit;
716     diff = value - old_limit;
717     ap_daemons_limit = value;
718     if(diff > 0) {
719       ap_max_daemons_limit = value;
720       idle_spawn_rate = diff;
721     } else if(diff < 0) {
722       idle_spawn_rate = 1;
723     }*/
724
```

```
725     //fprintf(stderr,"server_limit=%i, old_limit=%i ap_daemons_limit=%i!\n",
            server_limit, old_limit,ap_daemons_limit);
726     //fflush(stderr);
727   }
728
729   AP_DECLARE(void) ap_set_keep_alive_timeout(int value)
730   {
731     apr_socket_t *temp_socket;
732
733     ap_server_conf->keep_alive_timeout = apr_time_from_sec(value);
734     ap_scoreboard_image->global->keep_alive_timeout =
            ap_server_conf->keep_alive_timeout;
735
736     ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Server Rec
            Address (set): %i", ap_server_conf);
737     ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Keep Alive
            Timeout (set): %i", apr_time_sec(ap_server_conf->keep_alive_timeout));
738
739    /*
740     int index;
741     for (index = 0; index < ap_max_daemons_limit; ++index) {
742       if (ap_scoreboard_image->servers[index][0].status != SERVER_DEAD) {
743         temp_socket = ap_scoreboard_image->servers[index][0].sb_socket;
744         if(temp_socket != NULL) {
745       ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Keep Alive
              Timeout: %i", temp_socket->timeout);
746       temp_socket->timeout = apr_time_from_sec(value);
747
748         }
749       }
750     }*/
751     //fprintf(stderr,"AFTER server address %i timeout %i\n",
            ap_server_conf,ap_server_conf->keep_alive_timeout);
752     //ap_server_conf->keep_alive_timeout = apr_time_from_sec(value);
753   }
754
755   AP_DECLARE(void) ap_set_backlog(int value)
756   {
757     ap_listen_rec *lr;
758     for (lr = ap_listeners; lr; lr = lr->next) {
759       if(listen(lr->sd->socketdes, value) < 0) {
760             ap_log_error(APLOG_MARK, APLOG_CRIT, 0, ap_server_conf, "error
                setting backlog to %i: %s", value, strerror(errno));
761         } else {
762         ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "backlog set
              to %i", value);
763       }
764     }
765   }
766
767   AP_DECLARE(void) ap_update_child_rounds(ap_sb_handle_t *sbh)
768   {
769     worker_score *ws;
770     ws = &ap_scoreboard_image->servers[sbh->child_num][sbh->thread_num];
771     apr_proc_mutex_lock(ap_scoreboard_image->global->child_info.mutex);
772       ws->rounds++;
773       if(ap_scoreboard_image->global->child_info.rounds == 0) {
774         ap_scoreboard_image->global->child_info.rounds = ws->rounds;
```

```
775       } else {
776         // Update the request_info struct
777         apr_proc_mutex_lock(ap_scoreboard_image->global->request_info.mutex);
778           ap_scoreboard_image->global->request_info.keepalives -=
                ws->keepalives;
779           ws->keepalives = 0;
780           if(ap_scoreboard_image->global->min_keepalives_child ==
                sbh->child_num) {
781             ap_scoreboard_image->global->request_info.min_keepalives = 0;
782             ap_scoreboard_image->global->min_keepalives_child = -1;
783           }
784
785           if(ap_scoreboard_image->global->max_keepalives_child ==
                sbh->child_num) {
786             ap_scoreboard_image->global->request_info.max_keepalives = 0;
787             ap_scoreboard_image->global->max_keepalives_child = -1;
788           }
789         apr_proc_mutex_unlock(ap_scoreboard_image->global->request_info.mutex);
790       }
791
792       ap_scoreboard_image->global->child_info.rounds++;
793       ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Rounds updated
              to: %i", ap_scoreboard_image->global->child_info.rounds);
794
795       if(ws->rounds <= ap_scoreboard_image->global->child_info.min_rounds ||
             ap_scoreboard_image->global->min_rounds_child == sbh->child_num ||
             ap_scoreboard_image->global->min_rounds_child == -1)
796       {
797         ap_scoreboard_image->global->child_info.min_rounds = ws->rounds;
798         ap_scoreboard_image->global->min_rounds_child = sbh->child_num;
799         ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Min rounds
                updated to: %i by child: %i",
                ap_scoreboard_image->global->child_info.min_rounds,
                ap_scoreboard_image->global->min_rounds_child);
800       }
801
802       if(ws->rounds >= ap_scoreboard_image->global->child_info.max_rounds ||
             ap_scoreboard_image->global->min_rounds_child == -1)
803       {
804         ap_scoreboard_image->global->child_info.max_rounds = ws->rounds;
805         ap_scoreboard_image->global->max_rounds_child = sbh->child_num;
806         ap_log_error(APLOG_MARK, APLOG_DEBUG, 0, ap_server_conf, "Max rounds
                updated to: %i by child: %i",
                ap_scoreboard_image->global->child_info.max_rounds,
                ap_scoreboard_image->global->max_rounds_child);
807
808       }
809     apr_proc_mutex_unlock(ap_scoreboard_image->global->child_info.mutex);
810   }
811
812   AP_DECLARE(void) ap_get_control_logdata(struct control_logdata *data)
813   {
814     // Get data from the request_info struct
815     apr_proc_mutex_lock(ap_scoreboard_image->global->request_info.mutex);
816       data->requests_time =
              ap_scoreboard_image->global->request_info.requests_time;
817       data->nbr_of_requests =
              ap_scoreboard_image->global->request_info.nbr_of_requests;
```

```
818       data->rtt = ap_scoreboard_image->global->request_info.rtt;
819       data->rtt_var = ap_scoreboard_image->global->request_info.rtt_var;
820       data->keepalives = ap_scoreboard_image->global->request_info.keepalives;
821       data->min_keepalives =
              ap_scoreboard_image->global->request_info.min_keepalives;
822       data->max_keepalives =
              ap_scoreboard_image->global->request_info.max_keepalives;
823       ap_scoreboard_image->global->request_info.requests_time = 0;
824       ap_scoreboard_image->global->request_info.nbr_of_requests = 0;
825       ap_scoreboard_image->global->request_info.rtt = 0;
826       ap_scoreboard_image->global->request_info.rtt_var = 0;
827     apr_proc_mutex_unlock(ap_scoreboard_image->global->request_info.mutex);
828
829     // Get data from the child_info struct
830     apr_proc_mutex_lock(ap_scoreboard_image->global->child_info.mutex);
831       data->min_rounds = ap_scoreboard_image->global->child_info.min_rounds;
832       data->max_rounds = ap_scoreboard_image->global->child_info.max_rounds;
833       data->rounds = ap_scoreboard_image->global->child_info.rounds;
834     apr_proc_mutex_unlock(ap_scoreboard_image->global->child_info.mutex);
835
836     // Get data from the running_info struct
837     apr_proc_mutex_lock(ap_scoreboard_image->global->running_info.mutex);
838       data->running = ap_scoreboard_image->global->running_info.running;
839     apr_proc_mutex_unlock(ap_scoreboard_image->global->running_info.mutex);
840
841     // Get data from the running_idle_info struct
842     apr_proc_mutex_lock(ap_scoreboard_image->global->running_idle_info.mutex);
843       data->running_idle =
              ap_scoreboard_image->global->running_idle_info.running_idle;
844     apr_proc_mutex_unlock(ap_scoreboard_image->global->running_idle_info.mutex);
845   }
846   /*
847   AP_DECLARE(int) ap_is_new_test()
848   {
849     if(ap_scoreboard_image->global->new_test == 1) {
850       ap_scoreboard_image->global->new_test = 0;
851       return 1;
852     }
853     return 0;
854   }*/
855
856   AP_DECLARE(int) ap_get_stop_code()
857   {
858     int code = ap_scoreboard_image->global->test_stop_code;
859     //ap_scoreboard_image->global->test_stop_code = 0;
860     return code;
861   }
862   #endif
```

B.4 http_request.c

```
225   void ap_process_request(request_rec *r)
226   {
227       int access_status;
228
229       /* Give quick handlers a shot at serving the request on the fast
230        * path, bypassing all of the other Apache hooks.
231        *
```

```
232         * This hook was added to enable serving files out of a URI keyed
233         * content cache ( e.g., Mike Abbott's Quick Shortcut Cache,
234         * described here: http://oss.sgi.com/projects/apache/mod_qsc.html )
235         *
236         * It may have other uses as well, such as routing requests directly to
237         * content handlers that have the ability to grok HTTP and do their
238         * own access checking, etc (e.g. servlet engines).
239         *
240         * Use this hook with extreme care and only if you know what you are
241         * doing.
242         */
243        if (ap_extended_status)
244    #ifdef ELCONTROL
245            ap_time_process_request(r, START_PREQUEST);
246    #else
247            ap_time_process_request(r->connection->sbh, START_PREQUEST);
248    #endif
249        access_status = ap_run_quick_handler(r, 0);  /* Not a look-up request */
250        if (access_status == DECLINED) {
251            access_status = ap_process_request_internal(r);
252            if (access_status == OK) {
253                access_status = ap_invoke_handler(r);
254            }
255        }
256
257        if (access_status == DONE) {
258            /* e.g., something not in storage like TRACE */
259            access_status = OK;
260        }
261
262        if (access_status == OK) {
263            ap_finalize_request_protocol(r);
264        }
265        else {
266            ap_die(access_status, r);
267        }
268
269        /*
270         * We want to flush the last packet if this isn't a pipelining
              connection
271         * *before* we start into logging.  Suppose that the logging causes a
              DNS
272         * lookup to occur, which may have a high latency.  If we hold off on
273         * this packet, then it'll appear like the link is stalled when really
274         * it's the application that's stalled.
275         */
276        check_pipeline_flush(r);
277        ap_update_child_status(r->connection->sbh, SERVER_BUSY_LOG, r);
278        ap_run_log_transaction(r);
279        if (ap_extended_status)
280    #ifdef ELCONTROL
281            ap_time_process_request(r, STOP_PREQUEST);
282    #else
283        ap_time_process_request(r->connection->sbh, STOP_PREQUEST);
284    #endif
285    }
```

B.5 logger.h

```
1  #ifndef LOGGER_RUNNING_H
2  #define LOGGER_RUNNING_H
3
4  #include <stdio.h>
5  #include <pthread.h>
6  #include <time.h>
7  #include <unistd.h>
8  #include <sys/stat.h>
9  #include <sys/time.h>
10 #include <sys/types.h>
11 #include "httpd.h"
12 #include "http_config.h"
13 #include "http_log.h"
14 #include "scoreboard.h"
15
16 #define NBR_OF_FILES 26
17
18 #define STARTENDTIME 0
19 #define CPU_LOAD 1
20 #define MEM_ACTUAL_FREE 2
21 #define SERVICE_TIME 3
22 #define SERVED_REQUESTS 4
23 #define RUNNING 5
24 #define RUNNING_IDLE 6
25 #define SOCKSTAT_ALLOC 7
26 #define SOCKSTAT_MEM 8
27 #define SOCKSTAT_TW 9
28 #define RTT 10
29 #define RTT_VAR 11
30 #define SYN_FAILED 12
31 #define LISTENOVERFLOWS 13
32 #define KEEPALIVES 14
33 #define MIN_KEEPALIVES 15
34 #define MAX_KEEPALIVES 16
35 #define CHILD_ROUNDS 17
36 #define MIN_CHILD_ROUNDS 18
37 #define MAX_CHILD_ROUNDS 19
38 #define SOCKSTAT_ORPHAN 20
39 #define CPU_IO_WAIT 21
40 #define CPU_USER 22
41 #define CPU_SYSTEM 23
42 #define CPU_HIRQ 24
43 #define MEM_FREE 25
44
45 //global variables for the open_test_files and close_test_files functions
46 FILE *logger_files[NBR_OF_FILES];
47
48
49 char logger_path[50];
50
51 //struct used for the ap_get_socket_stat function
52
53 struct logger_sockstat {
54   int alloc;
55   int mem;
56   int orphan;
```

```
57    int tw;
58  };
59
60  //struct used for the ap_get_netstat function
61
62  struct logger_netstat {
63    int syn_failed;
64    int listen_overflows;
65  };
66
67  //struct used for the ap_calc_mem_usage function
68
69  struct logger_mem {
70    int actual_usage;
71    int usage;
72    int actual_free;
73    int free;
74  };
75
76  //global variables for the ap_calc_cpu_usage function
77  int logger_previous_total;
78  int logger_previous_idle;
79  int logger_previous_iowait;
80  int logger_previous_hirq;
81  int logger_previous_user;
82  int logger_previous_system;
83
84  //struct used for the ap_calc_cpu_usage function
85
86  struct logger_cpu {
87    float usage;
88    float iowait;
89    float hirq;
90    float user;
91    float system;
92  };
93
94  struct control_logdata last_read_control_logdata;
95  int logger_running;
96
97
98  //functions
99  AP_DECLARE(void) ap_calc_cpu_usage(struct logger_cpu *cpu);
100 AP_DECLARE(void) ap_calc_mem_usage(struct logger_mem *mem);
101 AP_DECLARE(void) ap_get_netstat(struct logger_netstat* netstat);
102 AP_DECLARE(void) ap_get_sockstat(struct logger_sockstat* sockstat);
103 void log_data();
104 void open_test_files(int want_running, int test_number);
105 void close_test_files();
106 void create_dir(char * dir_name);
107 int timeval_diff(struct timeval* result,struct timeval* x,struct timeval*
        y);
108
109
110 #endif
```

B.6 logger.c

```
1  #include "logger.h"
2  char *logger_file_names[] = {
3  [STARTENDTIME] = "startendtime",
4  [CPU_LOAD] = "cpu_load",
5  [MEM_ACTUAL_FREE] = "mem_actual_free",
6  [SERVICE_TIME]= "service_time",
7  [SERVED_REQUESTS] = "served_requests",
8  [RUNNING] = "running",
9  [RUNNING_IDLE] = "running_idle",
10 [SOCKSTAT_ALLOC] = "sockstat_alloc",
11 [SOCKSTAT_MEM] = "sockstat_mem",
12 [SOCKSTAT_TW] = "sockstat_tw",
13 [RTT] = "rtt",
14 [RTT_VAR] = "rtt_var",
15 [SYN_FAILED] = "syn_failed",
16 [LISTENOVERFLOWS] = "listen_overflows",
17 [KEEPALIVES] = "keepalives",
18 [MIN_KEEPALIVES] = "min_keepalives",
19 [MAX_KEEPALIVES] = "max_keepalives",
20 [CHILD_ROUNDS] = "rounds",
21 [MIN_CHILD_ROUNDS] = "min_rounds",
22 [MAX_CHILD_ROUNDS] = "max_rounds",
23 [SOCKSTAT_ORPHAN] = "sockstat_orphan",
24 [CPU_IO_WAIT] = "cpu_io_wait",
25 [CPU_USER] = "cpu_user",
26 [CPU_SYSTEM] = "cpu_system",
27 [CPU_HIRQ] = "cpu_hirq",
28 [MEM_FREE] = "mem_free",
29 };
30
31 const char *base_logger_folder = "/home/d00el";
32
33 int logger_init()
34 {
35   int pid;
36   pid = fork();
37   if(!pid) {
38     log_data();
39     exit(0);
40   }
41   return 0;
42  }
43
44 void open_test_files(int want_running, int test_number)
45 {
46   int i;
47   char file_path[50];
48   char whole_path[50];
49
50   sprintf(logger_path, "%s/matlab/data/wr%d", base_logger_folder,
       want_running);
51   create_dir(logger_path);
52   sprintf(logger_path, "%s/matlab/data/wr%d/test%d", base_logger_folder,
       want_running, test_number);
53   create_dir(logger_path);
54
```

```
55    for(i = 0; i < NBR_OF_FILES; i++) {
56      memcpy(whole_path, logger_path, 50);
57      sprintf(file_path, "/%s.dat", logger_file_names[i]);
58      strcat(whole_path, file_path);
59      logger_files[i] = fopen(whole_path,"w");
60      if(i != STARTENDTIME) {
61        fprintf(logger_files[i], "%s = [", logger_file_names[i]);
62      }
63    }
64  }
65
66  void log_data()
67  {
68    //fprintf(stderr, "Everything is just fine\n");
69    //fflush(stderr);
70
71    struct timeval start, stop, timeout, elapsed, interval;
72    struct control_logdata data;
73    struct logger_cpu cpu;
74    struct logger_mem mem;
75    int mem_load;
76    struct logger_sockstat sockstat;
77    struct logger_netstat netstat;
78    int want_running, test_number;
79
80    fd_set ef;
81
82    int t = 0;
83
84    interval.tv_sec = 1;
85    interval.tv_usec = 0;
86    FD_ZERO(&ef);
87
88    //fprintf(stderr, "before test\n");
89    //fflush(stderr);
90    //int ret = mkdir("matlab", 0777);
91
92    chdir(base_logger_folder);
93    create_dir("matlab");
94    create_dir("matlab/data");
95
96    apr_proc_mutex_lock(ap_scoreboard_image->global->logger_test.mutex);
97      want_running = ap_scoreboard_image->global->logger_test.want_running;
98      test_number = ap_scoreboard_image->global->logger_test.test_number;
99    apr_proc_mutex_unlock(ap_scoreboard_image->global->logger_test.mutex);
100
101   //fprintf(stderr, "after first new test arrived wr = %i, test = %i!!\n",
          want_running, test_number);
102   //fflush(stderr);
103
104   ap_set_want_running(want_running);
105   //fprintf(stderr, "After fetch new values: mcvalue = %d, test_number =
          %d\n", want_running, test_number);
106   //fflush(stderr);
107
108   open_test_files(want_running, test_number);
109
110   int run_tests = 1;
```

```
111     int stop_code = 0;
112     int iteration = 0;
113     logger_running = 1;
114
115     while(run_tests) {
116       if(stop_code = ap_get_stop_code()) {
117         logger_running = 0;
118         //fprintf(stderr, "End test!\n");
119       //fflush(stderr);
120         close_test_files();
121         //test_number++;
122
123         apr_proc_mutex_lock(ap_scoreboard_image->global->logger_test.mutex);
124           if(want_running !=
                ap_scoreboard_image->global->logger_test.want_running) {
125             want_running =
                  ap_scoreboard_image->global->logger_test.want_running;
126             ap_set_want_running(want_running);
127           }
128           test_number = ap_scoreboard_image->global->logger_test.test_number;
129         apr_proc_mutex_unlock(ap_scoreboard_image->global->logger_test.mutex);
130
131       if(ap_get_stop_code() == 2) {
132           run_tests = 0;
133           break;
134         }
135
136         //fprintf(stderr, "after new test arrived wr = %i, test = %i!!\n",
              want_running, test_number);
137         //fflush(stderr);
138
139         open_test_files(want_running, test_number);
140         iteration = 0;
141         logger_running = 1;
142       }
143
144       // Set start time for logging
145       gettimeofday(&start, 0);
146
147       if(iteration == 0) {
148         //fprintf(stderr, "Set new starttime %d + %d\n", start.tv_sec,
              start.tv_usec);
149       //fflush(stderr);
150         fprintf(logger_files[STARTENDTIME], "starttime = %d + %d/1000000;",
              start.tv_sec, start.tv_usec);
151       }
152
153       // Fetch data to log
154       ap_calc_cpu_usage(&cpu);
155       ap_calc_mem_usage(&mem);
156       ap_get_netstat(&netstat);
157       ap_get_sockstat(&sockstat);
158       ap_get_control_logdata(&data);
159       last_read_control_logdata = data;
160
161       // Log data to file
162       fprintf(logger_files[CPU_LOAD], "%.3f ", cpu.usage);
163       fprintf(logger_files[MEM_ACTUAL_FREE], "%i ", mem.actual_free);
```

```
164       fprintf(logger_files[SYN_FAILED], "%i ", netstat.syn_failed);
165       fprintf(logger_files[LISTENOVERFLOWS], "%i ", netstat.listen_overflows);
166       fprintf(logger_files[SERVICE_TIME], "%qd ", data.requests_time);
167       fprintf(logger_files[SERVED_REQUESTS], "%i ", data.nbr_of_requests);
168       fprintf(logger_files[RTT], "%i ", data.rtt);
169       fprintf(logger_files[RTT_VAR], "%i ", data.rtt_var);
170       fprintf(logger_files[RUNNING], "%i ", data.running);
171       fprintf(logger_files[RUNNING_IDLE], "%i ", data.running_idle);
172       fprintf(logger_files[KEEPALIVES], "%i ", data.keepalives);
173       fprintf(logger_files[MIN_KEEPALIVES], "%i ", data.min_keepalives);
174       fprintf(logger_files[MAX_KEEPALIVES], "%i ", data.max_keepalives);
175       fprintf(logger_files[CHILD_ROUNDS], "%i ", data.rounds);
176       fprintf(logger_files[MIN_CHILD_ROUNDS], "%i ", data.min_rounds);
177       fprintf(logger_files[MAX_CHILD_ROUNDS], "%i ", data.max_rounds);
178       fprintf(logger_files[SOCKSTAT_ALLOC], "%i ", sockstat.alloc);
179       fprintf(logger_files[SOCKSTAT_MEM], "%i ", sockstat.mem);
180       fprintf(logger_files[SOCKSTAT_TW], "%i ", sockstat.tw);
181       fprintf(logger_files[SOCKSTAT_ORPHAN], "%i ", sockstat.orphan);
182       fprintf(logger_files[SOCKSTAT_ORPHAN], "%i ", sockstat.orphan);
183       fprintf(logger_files[CPU_IO_WAIT], "%.3f ", cpu.iowait);
184       fprintf(logger_files[CPU_USER], "%.3f ", cpu.user);
185       fprintf(logger_files[CPU_SYSTEM], "%.3f ", cpu.system);
186       fprintf(logger_files[CPU_HIRQ], "%.3f ", cpu.hirq);
187       fprintf(logger_files[MEM_FREE], "%i ", mem.free);
188
189
190       //Set end time for logging
191       gettimeofday(&stop, 0);
192       timeval_diff(&elapsed, &stop, &start);
193       timeval_diff(&timeout, &interval, &elapsed);
194       //fprintf(stderr, "log data took %d sec %d usec\n", elapsed.tv_sec,
              elapsed.tv_usec);
195       //fflush(stderr);
196       select(1,&ef,NULL,NULL,&timeout);
197       iteration++;
198       /*if(iteration == 30) {
199         ap_set_backlog(0);
200       }*/
201     }
202   }
203
204   void close_test_files()
205   {
206     int i;
207     struct timeval endtime;
208
209     gettimeofday(&endtime, 0);
210
211     for(i = 0; i < NBR_OF_FILES; i++) {
212       if(i == STARTENDTIME) {
213         fprintf(logger_files[STARTENDTIME], "endtime = %d + %d/1000000;",
              endtime.tv_sec, endtime.tv_usec);
214       } else {
215         fprintf(logger_files[i], " ];", logger_file_names[i]);
216       }
217       fclose(logger_files[i]);
218     }
219   }
```

```
220
221 void create_dir(char * dir_name)
222 {
223   if(mkdir(dir_name, 0777) == -1 && errno != EEXIST) {
224     fprintf(stderr, "Error creating directory '%s' ERROR: %i\n", dir_name,
          errno);
225     fflush(stderr);
226     exit(0);
227   }
228 }
229
230 AP_DECLARE(void) ap_calc_cpu_usage(struct logger_cpu *cpu)
231 {
232   float usage=0,iowait_usage=0,system_usage=0,user_usage=0,hirq_usage=0;
233   char cpuid[6];
234   int user,nice,system,idle,iowait,hirq,sirq,steal,total;
235   FILE *fp;
236
237   fp = fopen("/proc/stat","r");
238
239   fscanf(fp,"%s %d %d %d %d %d %d %d
          %d",cpuid,&user,&nice,&system,&idle,&iowait,&hirq,&sirq,&steal);
240
241   fclose(fp);
242
243   total = user + nice + system + idle + iowait + hirq + sirq + steal;
244
245   usage = 1 - (float) (idle - logger_previous_idle) /
          (total-logger_previous_total);
246   iowait_usage = (float) (iowait - logger_previous_iowait) /
          (total-logger_previous_total);
247   system_usage = (float) (system - logger_previous_system) /
          (total-logger_previous_total);
248   user_usage = (float) (user - logger_previous_user) /
          (total-logger_previous_total);
249   hirq_usage = (float) (hirq - logger_previous_hirq) /
          (total-logger_previous_total);
250
251   logger_previous_total = total;
252   logger_previous_idle = idle;
253   logger_previous_iowait = iowait;
254   logger_previous_system = system;
255   logger_previous_user = user;
256   logger_previous_hirq = hirq;
257
258   cpu->usage = usage;
259   cpu->iowait = iowait_usage;
260   cpu->system = system_usage;
261   cpu->user = user_usage;
262   cpu->hirq = hirq_usage;
263 }
264
265 AP_DECLARE(void) ap_calc_mem_usage(struct logger_mem *mem)
266 {
267   int freemem, totalmem, usage,buffers,cached,actual_free;
268   FILE *fp;
269   fp = fopen("/proc/meminfo", "r");
```

```
270     fscanf(fp, "%*s %d %*s %*s %d %*s %*s %d %*s %*s
            %d",&totalmem,&freemem,&buffers,&cached);
271     fclose(fp);
272     //printf("TOTAL MEM: %i, FREE MEM: %i\n", totalmem, freemem);
273     actual_free = freemem + buffers + cached;
274     usage = totalmem - actual_free;
275     mem->actual_usage = usage;
276     mem->usage = totalmem - freemem;
277     mem->actual_free = actual_free;
278     mem->free = freemem;
279   }
280
281   AP_DECLARE(void) ap_get_sockstat(struct logger_sockstat *sockstat)
282   {
283     int alloc, mem, tw, orphan;
284     FILE *fp;
285       fp = fopen("/proc/net/sockstat", "r");
286       fscanf(fp, "%*s %*s %*s %*s %*s %*s %*s %d %*s %d %*s %d %*s
              %d",&orphan, &tw, &alloc,&mem);
287       fclose(fp);
288       sockstat->alloc = alloc;
289       sockstat->mem = mem;
290       sockstat->tw = tw;
291       sockstat->orphan = orphan;
292   }
293
294   AP_DECLARE(void) ap_get_netstat(struct logger_netstat *netstat)
295   {
296     int i, syn_failed, listen_overflows;
297     FILE *fp;
298     fp = fopen("/proc/net/netstat", "r");
299       for( i = 0; i < 69; i++) {
300         fscanf(fp, "%*s");
301       }
302       fscanf(fp, "%d",&syn_failed);
303       for( i = 0; i < 16; i++) {
304         fscanf(fp, "%*s");
305       }
306       fscanf(fp, "%d",&listen_overflows);
307       fclose(fp);
308       netstat->syn_failed = syn_failed;
309       netstat->listen_overflows = listen_overflows;
310   }
311
312   // carry out x - y where x and y are of the type timeval
313   int timeval_diff (struct timeval *result, struct timeval *x, struct timeval
          *y)
314   {
315     //printf("Ysec: %i, Yusec: %i\n", y->tv_sec, y->tv_usec);
316     /* Perform the carry for the later subtraction by updating y. */
317     if (x->tv_usec < y->tv_usec) {
318       int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
319       y->tv_usec -= 1000000 * nsec;
320       y->tv_sec += nsec;
321     }
322     if (x->tv_usec - y->tv_usec > 1000000) {
323       int nsec = (x->tv_usec - y->tv_usec) / 1000000;
324       y->tv_usec += 1000000 * nsec;
```

```
325       y->tv_sec -= nsec;
326     }
327
328     /* Compute the time remaining to wait.
329        tv_usec is certainly positive. */
330     result->tv_sec = x->tv_sec - y->tv_sec;
331     result->tv_usec = x->tv_usec - y->tv_usec;
332     //printf("Ysec: %i, Yusec: %i\n", y->tv_sec, y->tv_usec);
333     /* Return 1 if result is negative. */
334     return x->tv_sec < y->tv_sec;
335   }
```

B.7 controller.c

```
1  #include "logger.h"
2
3  int controller_init()
4  {
5    int pid;
6    pid = fork();
7    if(!pid) {
8      control_loop();
9      exit(0);
10   }
11   return 0;
12  }
13
14 void control_loop()
15 {
16   struct timeval start, stop, timeout, elapsed, interval;
17   struct control_logdata data;
18   //struct logger_cpu cpu;
19   //struct logger_mem mem;
20   //int mem_load;
21   //struct logger_sockstat sockstat;
22   //struct logger_netstat netstat;
23   int new_running;
24
25   fd_set ef;
26
27   interval.tv_sec = 1;
28   interval.tv_usec = 0;
29   FD_ZERO(&ef);
30
31   int iteration = 0;
32   int run_controller = 1;
33   int min_idle_processes = 32;
34   int max_idle_processes = 64;
35
36   while(run_controller) {
37     // Set start time for control
38     gettimeofday(&start, 0);
39
40     // Fetch data to log
41     //ap_calc_cpu_usage(&cpu);
42     //ap_calc_mem_usage(&mem);
43     //ap_get_netstat(&netstat);
44     //ap_get_sockstat(&sockstat);
```

```
45      if(logger_running)
46        data = last_read_control_logdata;
47      else
48        ap_get_control_logdata(&data);
49      if(data.running_idle > max_idle_processes) {
50        new_running = data.running - (data.running_idle - max_idle_processes);
51        ap_set_want_running(new_running);
52      } else if(data.running_idle < min_idle_processes) {
53        new_running = data.running + (min_idle_processes - data.running_idle);
54        ap_set_want_running(new_running);
55      }
56
57      //Set end time for control
58      gettimeofday(&stop, 0);
59      timeval_diff(&elapsed, &stop, &start);
60      timeval_diff(&timeout, &interval, &elapsed);
61
62      select(1,&ef,NULL,NULL,&timeout);
63      iteration++;
64      /*if(iteration == 30) {
65        ap_set_backlog(0);
66      }*/
67    }
68  }
```

Bibliography

[1] Statistics Sweden. Statistics Sweden. http://www.scb.se/templates/tableOrChart____187901.asp[Online; accessed 16-Jan-2008].

[2] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons Inc, 2004.

[3] Jeffrey C. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical report, Palo Alto, CA, DEC Western Research Laboratory, October 1995.

[4] Mikael Andersson. *Overload Control and Performance Evaluation of Web Servers*. PhD thesis, Department of Electrical and Information Technology, Lund Institute of Technology, 2007.

[5] Netcraft. Web Server Survey Archives, 2007. http://www.netcraft.com/survey [Online; accessed 01-June-2007].

[6] The Apache Software Foundation. The Apache HTTP Server Project.

[7] open source. lighttpd, 2007. http://www.lighttpd.net [Online; accessed 04-June-2007].

[8] Yixin Diao, Neha Gandhi, Joseph L. Hellerstein, Sujay Parekh, and Dawn M. Tilbury. Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics With Application to the Apache Web Server. Technical Report Network Operations and Management, IEEE/IFIP, April 2002.

[9] Anders Robertsson, Björn Wittenmark, Maria Kihl, and Mikael Andersson. Admission Control Web Server Systems - Design and Experimental Evaluation. In *IEEE conference on decision and control (CDC), Paradise Island, Bahamas*, 2004.

[10] Mikael Andersson, Maria Kihl, and Anders Robertsson. Modeling and Design of Admission Control Mechanism for Web Servers using Non-linear Control Theory. In *ITCom 2003, Orlando*, 2003.

[11] Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, and Oliver Schmidt. The Apache Modelling Project. Technical Report ISRN LUTFD2/TFRT--3239--SE, Hasso-Plattner-Institue for Software Systems Engineering, July 2004.

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall PTR, 1988.

[13] Free Software Foundation. The GNU C library, 2007. http://www.gnu.org/software/libc [Online; accessed 22-May-2007].

[14] Wikipedia. Apache Portable Runtime — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Apache_Portable_Runtime&oldid=117004634 [Online; accessed 13-April-2007].

[15] The Apache Software Foundation. Apache Portable Runtime Project. http://apr.apache.org [Online; accessed 13-April-2007].

[16] Peter Wainwright. *Pro Apache, Third Edition.* Apress, 2004.

[17] Lincoln Stein and Doug MacEachern. *Writing Apache Modules with Perl and C.* O'Reilly & Associates, Inc., 1999.

[18] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts, Fifth Edition.* John Wiley & Sons, 1999.

[19] Behrouz A. Forouzan. *Data Communications and Networking, 2nd edition.* McGraw-Hill, 2001.

[20] Pete Loshin. *TCP/IP Clearly Explained, 3rd edition.* Academic Press, 1999.

[21] Wikipedia. Internet protocol suite — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Internet_protocol_suite&oldid=121762712 [Online; accessed 14-April-2007].

[22] Pasi Saolahti. Linux TCP. Technical report, Nokia Research Center, October 2002.

[23] Glenn Herrin. Linux IP Networking, A Guide to the Implementation and Modification of the Linux Protocol Stack. Technical Report 1.0, Computer Science Department, University of New Hampshire, May 2000.

[24] Binh Nguyen. Linux Filesystem Hierarchy, 2007. http://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/index.html [Online; accessed 12-Mars-2007].

[25] Daniel A. Menascé and Virgilio A. F. Almeida. *Capacity Planning for Web Services.* Prentice Hall PTR, 2002.

[26] K. Park and W. Willinger. Self-similar network traffic: An overview. Wiley Interscience, 1999., 1999.

[27] Paul Robert Barford. Modeling, Measurement and Performance of World Wide Web Transactions. Technical report, University of Illinois, Urbana-Champaign, oct 2001.

[28] Karl-Erik Årzén. *Real-Time Control Systems.* Department of Automatic Control Lund Institute of Technology, 2003.

[29] Mikael Andersson, Martin Höst, Jianhua Cao, Christian Nyberg, and Maria Kihl. Content Adaption Schemes for Web Servers in Crisis Situations. Submitted to the ACM Transactions on Internet Technology, oct 2007.

[30] The MathWorks. Matlab. http://www.mathworks.com/products/matlab/.

[31] Sysstat utilities home page. http://perso.orange.fr/sebastien.godard/index.html [Online; accessed 13-April-2007].

[32] Rik van Riel. Measuring resource demand on linux. Linux Symposium, 2006.

[33] The Linux Documentation Projec. Linux-FAQ, troubleshooting. http://tldp.org/FAQ/Linux-FAQ/troubleshooting.html#free-memory-keeps-shrinking [Online; accessed 16-Mars-2007].

[34] Anders Hagsten and Fredrik Neisler. Crisis request generator for internet servers. Master's thesis, Lund Institute of Technology, June 2006.

[35] The Apache Software Foundation. The apache jmeter, 2007. http://jakarta.apache.org/jmeter/index.html[Online; accessed 22-May-2007].

[36] Wikipedia. Lamp (software bundle) — wikipedia, the free encyclopedia, 2007. http://en.wikipedia.org/w/index.php?title=LAMP_%28software_bundle%29&oldid=137137182[Online; accessed 10-June-2007].