Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Analysis and evaluation of high performance web servers

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Albert Hidalgo Barea

DIRECTOR: Rubén González Blanco

SUPERVISOR: Roc Meseguer Pallarès

DATE: July 13 th 2011

Title: Analysis and evaluation of high performance web servers

Author: Albert Hidalgo Barea

Director: Rubén González Blanco

Supervisor: Roc Meseguer Pallarès

Date: July 13 th 2011

Overview

Web servers are a very important tool when providing users with requested content on the Internet. Usage of the Internet is growing day-by-day, making those software applications essential.

In the first part of the thesis, the web server world will be introduced to the reader, by giving a brief explanation of some of the available technologies as well as different dynamic protocols. Also, as there are different web servers available in the market, during this report it will be chosen the best performing ones. So, it will be presented a comparative chart between all of them in order to show the most important features of each one.

Defining the scenario and the test cases is mandatory. For this reason, it is described the used hardware and software used to perform those benchmarks. The hardware is maintained equal during the whole test process, in order to let web server's performance gaps to their internal architecture. Operating system and benchmarking tools are also described and given some examples. Furthermore, test cases are chosen to show some strengths and weakness of each web server, enabling us to compare the relative performance between them.

Finally, the last part of the report consists on presenting the obtained results during the benchmark process, as well as presenting some lessons learned during the curse of the whole thesis, summing-up with some conclusions.

INDEX

CHAPTER 1. INTRODUCTION

Since the introduction of the Internet network, people can communicate with each other around the world using different online tools. Also, it has been a driving force to allow people to use electronic commerce to buy different goods, as well as share media content. Nowadays, there are almost 7,000 million **[1]** people in the world, with an estimation of up to 2,000 million Internet users. In order to see the importance of this share, in the year 2000 it was only 360,000 Internet users, so there has been an increase of 445%.

A web server powers all the operations done in the Internet, which is in charge of serving the requested content. Moreover, the second important tool in order to view this content is a web browser, which will translate lines of codes in a more understanding texts and images. As in consumer software, web servers are very powerful applications installed into computers that process the different web page languages. Not all the web servers found in the market are equal, they differ in some architectural points as well as in their distribution model.

In May 2011, it was counted a total of almost 325 million Internet sites **[2]** around the world. Being the web servers an important tool to provide content in the Internet, it is important to highlight some sharing between them. There are different well-known web servers, such as Apache Httpd, Microsoft IIS, Nginx, Google Blog and Lighttpd. The first one, Apache is the most used for this task, with a share of 62%, far ahead from Microsoft's solution, with only 18%.

1.1. Project scope

The aim of this project is to evaluate and benchmark different high performance web servers found in the market. For this reason, it will be chosen different web servers, which fulfill some important features, all of them open source. Used hardware will remain equal for all the scenarios. It will be useful to compare their programming architectures which, at the end, will set the differences in performance.

It is important to comment the performance differences regarding used technologies, and point out which could be the best suited for the different test cases. Chosen test cases will show their performance in different areas, which are being exploited by day-to-day usage. The absolute performance is not as important as the relative performance between each web server, which will emphasize the differences in architecture as well as features. Furthermore, it will be benchmarked different dynamic web technologies in order to see performance limits of them, and how they behave during the course of the test.

1.2. Background and motivation

This report is in line of what the Technology and Architecture Coordination (TAC) department is doing inside Telefónica I+D (TID) **[3]**. TAC department is in charge of evaluating and prototyping, as well as give technological support for other initiatives inside TID. Supporting different areas inside Telefónica I+D, gives TAC department a horizontal vision of all the current deployed software inside the company (see **Fig. 1.1**). In order to give support, it is needed to understand what the initiative is doing, which is the software that they are currently using or planning to use and, finally, give some initial prototype.

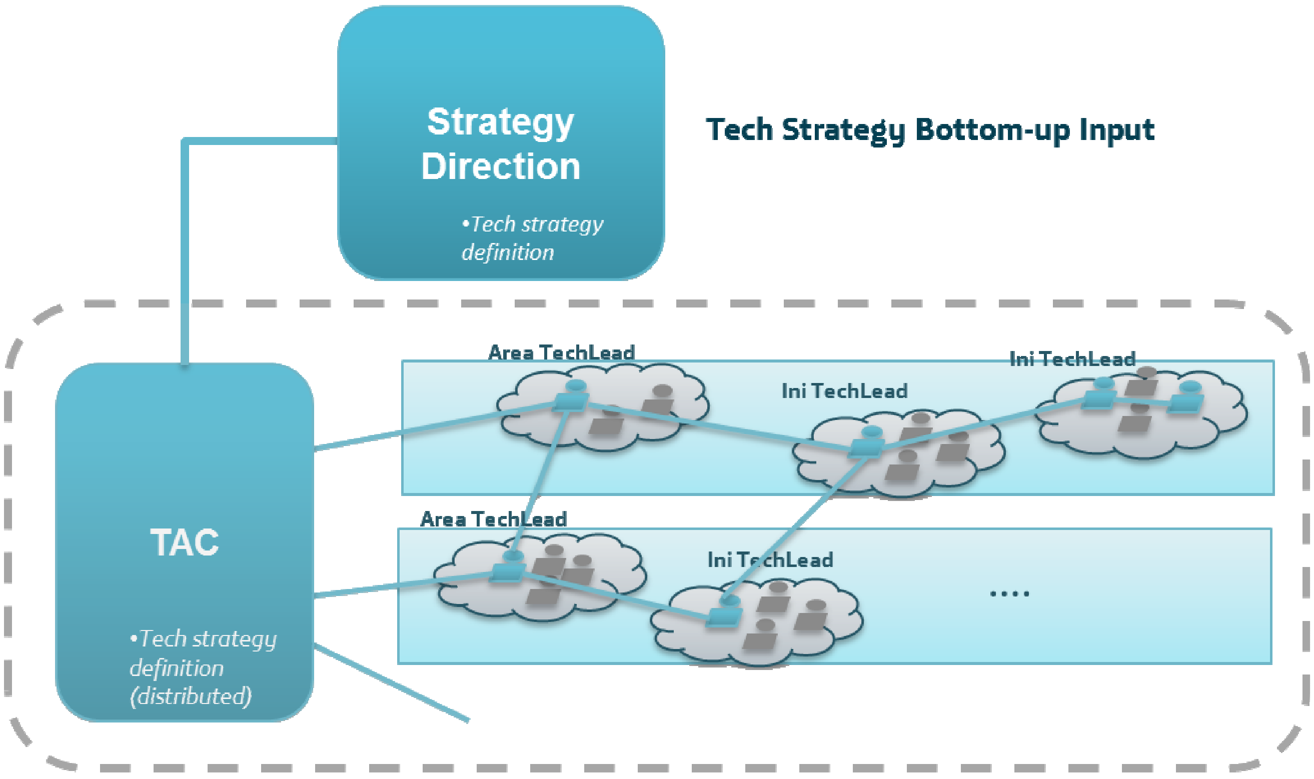
Strategy Direction
•Tech strategy definition
Tech Strategy Bottom-up Input
TAC
•Tech strategy definition (distributed)
Area TechLead
Ini TechLead
Ini TechLead
Area TechLead
Ini TechLead
....

Fig. 1.1: Technology and Architecture Coordination department

So, evaluating web servers, which are currently used in different Internet service projects, is an interesting report. This report could be useful as a master thesis, but also as an introductory point or reference document for deciding the use of some web server regarding their features.

1.3. Report organization

This master thesis is divided in five chapters, in which of them will be introduced different aspects of this project. The first chapter is meant to introduce the reader into the report, trying to highlight the interest of doing an analysis and evaluation of high performance web servers. As stated in the introduction, the evolution of the Internet networks makes necessary to have high performance systems in order to serve all the possible requests. For this reason, it is needed high performance hardware, but also high performance applications, which at the end, will be in charge of serving those answers.

The second chapter consists in presenting different web servers as well as some needed technology and architecture concepts. For this reason, it is given a brief introduction to HTTP as well as other web content technologies that makes possible to serve content to Internet users. Afterwards, it will be presented the chosen web servers for this report, and some architecture concepts to take into account during the results chapter.

The next chapter will define the testbed followed to benchmark each web server. It is important to define as good as possible the test cases as well as the whole scenario, in order to take into account all the variables. It will be presented some hardware and operating system key information, in order to expect some performance results. Those given characteristics will not change during the entire benchmark process, giving the opportunity to extract some relative results. There will be different oriented test cases, in order to see strengths and weaknesses of each web server. Finally, a more realistic test will be defined, which will allow to present more real performance results.

Chapter four brings the obtained results. This is one of the most important chapters of the report, as it will show all the performance results obtained during the different test cases defined previously. Also, it will be possible to extract some conclusions derived from each test, and at the end, give a recommendation of software to use regarding the needs of each scenario. Finally, it will be presented some lessons learned during the execution of this master thesis as well as some conclusions about the project.

CHAPTER 2. WEB SERVERS

The use of Internet in our day-to-day duties is becoming very common. Nowadays, the usage of Internet is increasing as more people have the opportunity to access it. And, this is thanks to the deployment of many different services, such as online shopping, information consultant or media related.

Internet was developed by DARPA (department of Defense of the United States of America) in the 60's **[36]**. But, its purpose was to be a military network where exchange confidential information. It did not exploited to be a communication tool since the end of 1990, when Tim Berners-Lee **[4]** (while he was working for the CERN **[5]**) begun the development of the first browser, called "WorldWideWeb", which has been the seed for further browsers and web development. Internet has evolved to become a very important communication way, displacing in some cases more traditional ways of communication such as readable, radio or television.

Programming for the web is like programming for the computer environment, as latest features introduced in the web are very similar to those found in the desktop. There are different programming languages used in the web (e.g., PHP, Java, Python, etc.), and all of them belong to high level programming languages. HTML language, which is used in all the web pages, is a markup language **[35]** and therefore, is not considered as a programming language. Moreover, accessing content in the Internet can be done by different means, such as HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol), SSH, etc. Although, the most used in order to surf the web is HTTP.

2.1. Web languages

A programming language is an artificial language created to give the possibility to interact between humans and machines. In the history of the computers there has been a lot of them, some of them focused in different areas. The increase of web usage gives the opportunity to create new features and also use even more dynamic languages, as some features are not anymore reserved for desktop applications. There are different categories to classify the languages used in the web. The goal of this report is not to describe all the categories and types of programming languages, but it is useful to have an idea of what is being used in the computer programming world.

2.1.1. Programming languages classification

2.1.1.1. Low level programming languages

This type of language provides little or any abstraction from computer instructions, being as close as possible to the hardware. The code can be run directly to the processor without using any compiler. Low level programming languages can be made to run very fast and with a low memory footprint; but it also has some important drawbacks. It must be run on specific machines, as not all the processors have the same instructions sets. Also, it is more difficult to understand when programming and debugging.

Web development could not be done using this approach, as developers would need to create specific programs for each hardware available in the market. Also, it would not be possible to deploy web applications, because the time spent to create and maintain those applications would be high.

2.1.1.2. High level programming languages

A high level programming language is the programming language that offers high abstraction to the instruction set that understands the processor. Normally, these programming languages are programmed using more natural language, which gets closer with the human communication. Also, they do not show processor properties, as it needs a compiler in order to convert the script into assembly code. Eventually, any programming language must be converted to machine instructions to be executed. In comparison with low level programming language, high-level ones have higher portability, as maybe they are set to work in different platforms with different hardware. This property is part of the compiler, which sets the different parameters to run into the hardware.

There are different executing types of high level programming languages. The first one is compiled languages. Those languages are written in a program and before being executed, they must be compiled. Compilation process converts from script to machine instruction set, which later would be executed. The result file is platform specific, as it contains the instruction set of only one platform. This is given by the compiler, which will work only in a given platform. As, for instance, running a C compiled program in Windows and in Linux machine is not possible. Eventually, the web developer would create its web application and deploy it always in the same platform (for instance, Linux web servers supporting different Linux operating systems' distributions).

The second one is compiled languages on virtual machines. This case is similar to the previous one, but with the exception of adding a virtual machine between the script and the operating system. Virtual machine is in charge of converting the script into the instruction set that the hardware will understand. It gives portability to the script, as it could be executed in any similar virtual machine. In order to ensure that the script can run, there must be a virtual machine for each environment. One virtual machine that is widely used is Java. It has a virtual machine for almost any operating system, allowing portability of all Java programs and Java web applications. But, the main drawback of this execution type is the performance. Introducing a new layer between the script and the operating system decreases the performance of the system, although it introduces portability.

In third place are interpreted programming languages. Those languages run indirectly into the operating system. Interpreting language gives some flexibility against compiled languages. These are platform independent, as they only need the main library installed in the operating system to run. Some interpreted web languages are PHP, Javascript, Python, etc. Those languages are being used to provide dynamic content to web pages. But, those high level programming languages in order to run as a web application must be used together with a dynamic web content technology, which will act as a gateway for them.

2.1.2. Web content classification

2.1.2.1. Static content

Static web content consists in showing a web page as equally as it is stored into the web server. It does not offer any user interaction delivering the requested file without any change. Later on, when the web browser receives the file it displays it in a user-friendly way. Some examples of this static content classification can be HTML only web pages, as well as images. This type of content is equal for all the users viewing the web page, no offering personalized web navigation. But, on the other hand it is not computational expensive for the web server, as it does not need to process anything.

2.1.2.2. Dynamic content

Dynamic web content are those web pages which offer user interaction or are generated by the web server. In this case, the web server must offer some kind of language interpreter or compiler to process the requested web page file. Moreover, it can be configured to display different content to different users, allowing a more personalized web experience.

There are two different dynamic content web pages classification, regarding the dynamic content which is being used. The first one, are those web pages using client-side scripting and content creation. In this case, dynamic behavior occurs within the presentation layer in the client’s web browser, in response for some actions, such as mouse over a section, keyboard actions or timings. Some examples of client-side dynamic languages are JavaScript or ActionScript. Usually, this dynamic language is embedded into the HTML web page, and it is being processed by the web browser once it is showed.

Meanwhile, server-scripting and content creation are those programs being executed in the web server side and used to personalize the user’s navigation experience. The response web page offered by the web server is triggered by some HTML form post parameters, URL parameters, or simply by the type of web browser. Those dynamic content web pages are used with server-side languages such as PHP, Python, Java, etc. Those languages, as explained in the previous section, must be used with some compiler or interpreter, which will increase the web server generated load.

2.2. Dynamic web content technologies

Dynamic web pages are very important today. Internet growth and the introduction of new services like electronic commerce or media related, make use of dynamic web pages, which is an essential feature in today's web servers. The consequence of using such dynamic content is that web servers are getting more stressed, loading even more their hardware. At the end, this is translated in a performance penalty for the web server.

2.2.1. CGI

CGI stands for Common Gateway Interface **[10]** and it is a simple interface for running external programs, software or gateways from HTTP servers. It is standardized in the RFC 3875. So, CGI allows HTTP servers to share the request response. The web server is responsible for managing connection, data transfer, transport and network issues, whereas CGI script handles the application issues, such a data access and document processing.

Common Gateway Interface has been the de facto standard of processing dynamic contents in the HTTP web servers, but newer solutions increase its performance as well as security. It is possible to write CGI programs for most available programming languages, such as PHP, Python, C, Perl, etc. The working principle of CGI is as follows. When there is a request to a certain URL, the web server receives it and processes it. If the requested information matches some rules, which make them dynamic, passes it to the CGI software. The CGI software performs the instructions given in the requested script. Once the information is processed, the CGI program process the output in such a way that the web server will understand (for example, creating an HTML page). Finally, the web server will return the information to the user, once all this process has ended.

But CGI has also some drawbacks related to performance and security. Calling a CGI script means the creation of a new process. So each time that a CGI path is requested a new process must be created to address this request. Creating a new process has its own performance penalty. The operating system must allocate some memory for it, and also the load generated when changing processes affects the performance. To minimize this effect, it may be used complied languages which consume fewer resources compared to interpreted ones, but still is a slow method. There are some security issues when using CGI software. To run CGI scripts, the file must be executable, so you are letting someone to run a program in your server. Some hosting companies do not let users to run CGI scripts, for this reason CGI can be redirect to other machines. As scripts are executable, if they have malicious code it could be a big security issue.

2.2.2. FastCGI

FastCGI **[11]** is an evolution of the previous explained CGI protocol. This increasingly use of dynamic web pages has highlighted the performance limits of CGI. FastCGI tries to solve some of the problems explained in CGI, and after all get better performance.

Very large and interpreted applications may have a slow start. Also, initialization such as logging on to database or connections to remote machines may impact the performance. In contrast of CGI, FastCGI processes are persistent. After finishing a request, this process waits idle for a new request instead of finishing the process. This lowers the performance penalty of creating new processes in the operating system. When configuring FastCGI, there is the possibility to specify how many FastCGI processes to spawn. This option lets the user to distribute the load between processes, and finally, increase the performance of the web site by running multiple requests simultaneously.

FastCGI is language and server independent. It has different server APIs to bind it with different web servers, as well as development libraries for mayor programming languages such as C, C++, Perl, Python, PHP, etc. This interoperability between web servers and platforms allows an easier deployment. It is an open standard, so anyone can implement it or improve it. Finally, using FastCGI does not imply mayor changes to script programmer, because it is like programming for CGI. Security can also be increased as FastCGI can communicate over TCP/IP connections, which gives the possibility to run applications remotely from the web server. Apart from security, it can also provide some scalability, load balancing features and high availability. It implements new functionalities to support different application roles, such as responder (the basic FastCGI role, which is the same as CGI), filter (FastCGI filters the requested web server file before sending it), and authorizer (FastCGI program performs an access control for the request).

2.2.3. Servlet

A Servlet **[12]** is a Java class, which is used to extend the capabilities of servers that host applications accessed via a request-response programming model. Servlets are the response of Java programming language to CGI. Java Servlets are more efficient, portable and with higher performance than CGI technology. As said before in the CGI section, when a request is made to a CGI page, a new process is created to answer this request. The creation of a new process each time has a computational cost, and could it be that the process creation takes more time than the CGI execution. Servlets solve this, by not creating a new process each time a new request arrives. Servlet requests are handled by a separate thread, avoiding some of the problems of creating processes.

When simultaneous request to the same CGI script arrives to the web server, it loads and copies into system's memory the same script as many times as concurrency level indicates. However, Servlets create one new thread per new request, but there will be only one copy of the Servlet script in the system memory, which will be shared between all active threads. So, running only one instance of the Servlet reduces the memory usage and increases the performance of the web server. But sharing the address space makes them less robust, and forces the programmer to ensure the use thread safe features.

As a security concern, Servlets are executed within a restrictive environment. This environment is called sandbox. This model gives different levels of trust based on the source of the Servlet, giving open access to the server or limited access by a security manager. Java Servlets have good portability as they are written in Java. Java API is well standardized so any compiling platform with that API will accept the Servlet.

2.2.4. JSP

JavaServer Pages (JSP) **[13]** is a Java technology that enables the creation of dynamic web pages based on HTML or XML tags. Being part of Java programming language, JSP applications are feature rich and also platform independent. JSP technology uses XML-like tags to encapsulate the logic that generates the content for the page. Meanwhile, the application logic can reside inside the web server.

This separation of the logic and the content allows the programmer to do quick changes and also reuse the design. Being an extension of a Java Servlet technology makes JSP platform independent, as it can be executed in any Servlet compiling web server. One advantage of JSP against Servlets is that it is easier to write and edit HTML code than to do the same with Servlet instructions (as for instance *println*). Also, the separation of the logic and the content could lead to split of tasks. As said before, JSP uses HTML or XML like tags, apart from the HTML code of the web page. To add JSP functionalities to the web page, the function have to be introduced between <%= %> or <% %>. Inside these brackets it is possible to add directives or instructions that make reference to Java code, which will be interpreted and processed by the web server.

2.2.5. uWSGI

WSGI **[14]** is the Web Server Gateway Interface for Python programming language. It is a specification for web servers and application servers to communicate with web applications. It is a Python standard, described in detail in PEP 333 **[15]**. The goal of WSGI is to provide a relatively simple yet comprehensive interface capable of supporting most of interactions between a Web server and a Web framework.

The WSGI interface has two sides: the "server" or "gateway" side, and the "application" or "framework" side. The server side invokes a callable object that is provided by the application side. In addition to "pure" servers/gateways and applications/frameworks, it is also possible to create "middleware" components that implement both sides of this specification. WSGI is lower level than CGI, but in difference to CGI, WSGI does scale and can work in both multithreaded and multi process environments. WSGI is not CGI because it is between the web application and the webserver layer, which can be CGI, mod_python (or another module), FastCGI or a webserver that implements WSGI in its core.

uWSGI **[16]** is a fast, self-healing and developer/sysadmin-friendly application container server coded in C. It can be run in preforming mode, threaded, asynchronous/evented. It has some features such as low memory footprint, master process manager, UNIX and TCP socket support, etc.

2.3. HyperText Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) **[6]** is a stateless application-level protocol for distributed, collaborative, hypermedia information systems used in the World Wide Web. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. HTTP has been in use by the World Wide Web global information initiative since 1990.

Since 1990, there have been different HTTP specifications by the IETF. The first version was the HTTP/0.9 **[7]**, and was a simple protocol for raw data transfer across the Internet. HTTP/1.0 **[8]** improved the standard in 1996 by allowing messages to be in the format MIME-like messages. These messages contain information about the data that is being transferred. However, as usage of the Internet increases, the need to interact with proxies, content catching, use persistent connections or virtual hosts was not satisfied.

The latest version, HTTP/1.1 **[9]**, released in 1999 introduced some changes to the protocol. In HTTP/1.0, most implementations used a new connection for each request/response. Meanwhile, in HTTP/1.1 a connection may be used for one or more request/response exchanges, although connections may be closed for a variety of reasons. HTTP communication usually takes place over TCP/IP connections. The default port for the communication is TCP 80, but other ports can be used. Using a TCP port does not force to use always TCP architecture, only that presumes a reliable transport protocol. Any protocol that provides such guarantees of reliability could be used too.

HTTP protocol is a request/response protocol. The working principle is as follows. A client sends a request to the server in the form of a request method, in which includes the URI of the object, HTTP protocol version, MIME headers and client information. Next, the server answers it with a message, which includes the protocol version and a success or error code, followed by a MIME-like message containing server information, and possible entity-body content. As stated before, HTTP is a negotiation protocol. It is a mechanism to select the appropriate representation when answering a request from a client. Some of the negotiations include: used charset, content coding, transfer coding, media type, and language tags.

HTTP messages are composed by two parts, the headers and the body. The first one is where is placed all the information needed to display the transferred data, in which are delivered all the possible negotiations described before. Those parameters are read by the browser to prepare the data; not always will be available all the headers, so the order is not important. Finally, the body is the part of the message where the data of the message is. This data will be displayed with the given format as well as with all the parameters present in the header.

2.4. Web server selection

When visiting a web page in the Internet there must be a HTTP web server in order to serve that request. Web servers need to be able of processing this request and send the response to the different clients. As said previously in this chapter, there are differences between static content and dynamic, being more expensive (in terms of CPU processing) to execute the latest one.

In next sections of this chapter, the attention will be focused on selecting the web servers for this comparison, and explain their architecture and main features.

2.4.1. Available web servers

There are different providers of web servers in the market, some of them are free, meanwhile other are commercial. In this report, it will only be tested open source web servers. As said before, it will be chosen web servers that are highly customizable by the user and with enough interesting features.

2.4.1.1. Apache httpd

Apache httpd **[17]** is a collaborative project that involves people around the world to create a robust, commercial-grade, featureful and open source HTTP web server. The project is part of the Apache software foundation, which also creates and supports a bunch of different open source projects. Httpd is a very powerful web server and highly customizable by the final user. Is so extended and customizable that is included in a lot of Linux distributions, being a perfect starting point for anyone who wants to create a web site. Also, there is the possibility to download some bundles in which there is packed the Apache httpd web server, the PHP and MySQL libraries. Those packages enable the possibility to create a very powerful and dynamic web site without any complicate configurations. Other important aspect is that it is available for almost any platform in the market.

2.4.1.2. Lighttpd

Lighttpd **[18]** is a secure, speedier, compliant and flexible web server, designed and optimized for high performance environments. It has a low memory footprint compared to other web servers, so it is suitable for systems with limited resources of CPU and RAM memory. *Lighty* is a single-threaded, single-process, event-based, non-blocking-IO web server.

2.4.1.3. Cherokee

Cherokee **[19]** is a very fast, flexible and easy to configure Web Server. It supports the most used technologies in the web server front, like CGI, Fast CGI, SCGI, uWSGI, TLS/SSL encryption, load balancing, reverse HTTP proxy and much more. Cherokee is configured as a single process but with multithread support. One differentiate thing that includes Cherokee, is a user-friendly web interface in which is possible to configure the whole web server. It is very easy to set up the web server and tune it to get the better performance, as well as configure all supported technologies such as Fast CGI, uWSGI, etc.

2.4.1.4. Nginx

Nginx **[20]** is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. Nginx is one of a handful of servers written to address the C10K problem (**Ref. [21]**). Unlike traditional servers, Nginx doesn't rely on threads to handle requests. Instead, it uses a much more scalable event-driven (asynchronous) architecture. This architecture uses small, but more important, predictable amounts of memory under load. It is also possible to fork the web server in more than one process, in order to take profit of all the CPU power.

2.4.1.5. Mongrel2

Mongrel2 **[22]** is an application, language and network architecture agnostic web server that focuses on web applications using modern browser technologies. The term “language agnostic” means that Mongrel2 does not try to promote any language over any others. It only knows about HTTP requests, HTTP responses, and asynchronous messages. So, finally, it is independent of what the final user chooses to work with, and tries to run all of them. In order to properly do asynchronous, Mongrel2 uses ZeroMQ, which is a decentralized message-oriented-middleware. Using ZeroMQ lets Mongrel2 communicate with different languages, operate without following determinate network architecture, and do it with a very simple communication model and API.

2.4.1.6. Hiawatha

Hiawatha **[23]** is an open source web server with a focus on security. This results in a highly secure web server, in both code and features. Hiawatha can run on Linux, BSD, Mac OS X and Windows. It supports all kind of technological features like CGI, Fast CGI, keep-alive support, SSL and more features. Although it supports dynamic languages, it has been optimized and tested to run with PHP language. The adoption of Hiawatha is small, so the developer of the web server announced in March 2011, that there will be support for the users but the releases and the features introduced to Hiawatha will be reduced. Although this announcement, it is worth the try and see what has to offer.

2.4.1.7. Tomcat

Apache Tomcat **[24]** is an open source software implementation of the Java Servlet and JavaServer Pages technologies. Tomcat provides a pure Java HTTP web server environment for the java code to run. Servlet and JSP pages are dynamically loaded into the web server, so their performance will be lower than for static contents. Tomcat offers a high degree of customization as well as lot of functionalities. It is an application web server, meaning that is intended to serve web applications, not only web pages. Also, Tomcat gives the possibility of running static scripts as well as dynamic by means of CGI protocol. One big missing feature is the Fast CGI protocol, but Tomcat is focused in executing Servlets and JSP.

2.4.1.8. Yaws

Yaws **[25]** is a HTTP high performance 1.1 web server particularly well suited for dynamic-content web applications. Yaws is entirely written in Erlang and, furthermore, it is a multithreaded web server where one Erlang lightweight process is used to handle each client. The performance advantage of Yaws comes from the use of the different Erlang libraries, which can handle concurrent processes in an efficient way, using Erlang’s OTP.

2.4.2. Web servers feature comparison

After selecting which web server will be used in this report, it is needed to compile their features and expose them in a comparison table. **Table 2.1** shows the different features available for each one of the web servers in this report. All of them are based in the usage knowledge as well as information in their official web pages.

Table 2.1: Web server feature comparison.

	Apache httpd	**Lighttpd**	**Cherokee**	**Nginx**	**Mongrel2**	**Hiawatha**	**TomCat**	**Yaws**
Provider	Apache foundation	Lighttpd	Cherokee	Nginx	Mongrel	Hiawatha	Apache foundation	Yaws
Version	2.2.17	1.4.28	1.2.0	0.8.54	1.5	7.4	7.0.10	1.89
Build	October 2010	August 2010	February 2011	December 2010	January 2011	November 2010	March 2011	Sep. 2010
Request scheduling (see **section 2.4.3**)	Multi process, Multithread, Event driven (in beta mode)	Single process single thread event oriented Possibility of fork processes	Single process multi thread	Single process single thread event oriented Possibility of fork processes	Single process single thread event oriented	Single process multi thread	Single process multi thread	Single process multi thread
Kernel language	C	C	C	C	C	C	Java	Erlang
Supported OS	Windows, Linux, Mac OS X, BSD, Solaris	Windows, Linux, Mac OS X, BSD, Solaris	Windows, Linux, Mac OS X, BSD, Solaris	Windows, Linux, Mac OS X, BSD, Solaris	Linux	Windows, Linux, Mac OS X, BSD, Solaris	Windows, Linux, Mac OS X, BSD, Solaris	Windows, Linux, Mac OS X, BSD, Solaris
HTTPS	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
IPv6 support	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Modules	Yes	No	No	No	No	No	No	No
CGI	Yes	Yes	Yes	Not officially supported	No	Yes	Yes	Yes
FastCGI	Yes	Yes	Yes	Yes	No	Yes	No	Yes
uWSGI	Yes	Not officially supported	Yes	Yes	No	No	No	No
Java Servlet	No	No	No	No	No	No	Yes	No
JavaServer Pages	No	No	No	No	No	No	Yes	No
Ease of deployment	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Developer friendly	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Community	Very big	Big	Big	Big	Small	Small	Very big	Small
Open source	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

2.4.3. Architecture overview

As seen in **Table 2.1**, not all web servers run in the same way. There are three different options available. The first difference that can be seen, is that some of the web servers use a multi process model, where is possible to fork the main process; meanwhile, some others use a multi thread approach or event-driven architecture. All of them work in almost all scenarios, but there are differences that must be taken into account when deciding to work with one of the technologies.

2.4.3.1. Process oriented

A process is an executing instance of an application. Each process that is launched into the kernel of the operating system contains the program code to execute, as well as information about its state. Every process uses its own address space, and only interacts with other processes by means of the inter-process communication mechanisms, which is managed by the operating system. Switching between processes is very expensive for the processor, as it needs to move from one memory space to another. Therefore, this option has a performance penalty.

Usually, applications have a master process where is load the main functionality of the program. Later on, this main process may start new processes when it is needed to separate some functionality from the main program. As processes run separately from each other, in case of failure of one of them, only the affected process will fall, meanwhile the others will continue working. In web servers, if an Apache process crashes (for instance, a program error or buffer overrun) the remaining processes will continue working; and, eventually, create a new process to substitute the crashed one. On the other hand, if Cherokee process (take into account that it only runs a single process) crashes, the whole web server will stop. This is because there is not the possibility to spawn a new process without user’s intervention.

Apache httpd web server runs by default in a multi process environment. There is a master process that spawns as many as selected instances of Apache. Those new processes are created and running waiting for requests to process, it is a pool of processes. The user must configure in the httpd configuration file, the number of processes to create as well as the condition to start new ones.

Summing up, with this process approach, it can be said that this environment is more stable as each process run separately from the others, and usually there is a father process, which controls each child process. But there is a performance penalty associated to changing between processes in the operating system.

2.4.3.2. Multithread oriented

A thread is the basic unit to which the operating system allocates processor time. It is usually the result of the division of a program, allowing it to execute more than two tasks at the same time. As threads **[32] [33]** of a program are part of the program itself, they share some of the information such as address space, I/O operations or sockets. Threads require fewer overheads than creating new processes, because the system does not need to allocate new memory for them. Also, threads are more efficient because when changing from one thread to another there is no need of changing the whole information, as they share part of it.

Multithread works in single CPU as well as multiprocessor systems. In case of a single threaded process working in a single processor, if the main thread blocks, the whole process will freeze until it finishes. Moving to a multithread environment allows the application to perform more than one task at a time. On the other hand, in multiprocessor systems, having a multithreaded program allows the operating system to distribute the execution of each thread between the different available cores. This technique will increase the whole performance of the system.

But, there are some drawbacks when using threaded applications. The programmer must ensure that multiple threads are not working with the same set of data. This is called race condition. The programmer writes the way to execute threads in the code, but the operating system schedules when they will be launched. Another problem that appears when running threaded programs is the term thread safe. Some of the variables that are executed by the threads can be shared by more than one thread, which could provoke a conflict of data. If one thread crashes, it will crash the whole process, no matter how many threads this process has.

Multithread environments could perform better than multi process in some cases, but they demand a more accurate programming. In the case of Apache httpd web server, when installing is needed to specify which type of multi-processing module wants to be used, whether multi process based (called Prefork mode) or multithread based (called Worker mode). When selecting Worker mode, the user must specify the maximum threads allowed to each process, as well as the conditions to create a new processes.

Memory is also a finite resource in the system, and it is also a critical aspect to control when deploying a web server. Not only CPU load is important to measure the actual load of the server, but also the memory usage. Each process consumes a percentage of memory of the machine. Multithread applications share the same memory resources between threads, and this results in a lower memory footprint, compared with pure multi process applications. Eventually, this will affect the whole system's performance, as it will mark the number of processes that could be open.

2.4.3.3. Event-driven (asynchronous) oriented

Event-driven architecture (EDA) is a style of software architecture based on real time flows that detect, process, consumes, produces, and reacts to events. An event can be defined as a significant change of some kind of data or state. EDA has different components in order to process the generated events, such as a producer, consumer or listener, and a processor. This architecture is asynchronous and loosely coupled, as it is the event generator or producer who tells the listener that there has been a change. EDA could also be described as a push based architecture, as it needs to be notified when something changes, rather than polling the origin system that has generated the event. This kind of architecture is very useful also for message-oriented-middleware (MoM) systems, which need to process data from different components.

In web servers, implementing event-driven architecture means that one process handles all the events **[31] [32]**. This approach consists in a single process single threaded, which is able to process multiple connections using non-blocking I/O mechanisms. In order to know which connections are ready to write or read, are used some system directives such as *select*, *poll*, or *epoll*. Epoll **[34]** directive has a better performance than the previous ones, and it is recommended to use in high performance applications.

For network I/O operations it is possible to use non-blocking mechanisms present in the UNIX kernel. Using blocking directives, the running thread blocks until all data is transmitted by the operating system's kernel. Meanwhile, using non-blocking strategies is possible to send part of the requested data and the remaining data later on. Next, for each new event, is executed the corresponding event-handler, which will process the generated event. Those events could mean the acceptance of a new connection, the read or write of a HTTP connection or some logging feature. Using non-blocking directives is not always possible. Disk I/O is usually a blocking directive. Using this event-driven approach involves a single process single thread. When access to the disk is needed, the whole server stops until this I/O instruction is completed, decreasing the performance of the web server. It gets critical when there is a large amount of concurrent connections and they need access to the disk, as it will take longer than expected to finish.

In order to avoid disk-blocking issue, there is the possibility to run more than one process at once. This will help dealing with I/O blocking directives as well as working with multiprocessor schemas. However, this technique needs that each process must listen to a different socket, needing additional processing to balance requests between the different sockets. Lighttpd and Nginx work using this approach. By default, they are loaded using a single process single thread event-driven architecture, but to get better performance is advisable to spawn other processes. Apache has its own implementation of event-driven architecture; in which it is needed to specify the use of this multi-processing module when configuring. According to the official documentation of Apache httpd web server, the event multi-processing module is no longer experimental in version 2.4, but yet is a beta release.

2.5. Software stack

Summing-up this section, it is important to highlight the different layers that were explained before (see **Fig. 2.1**).

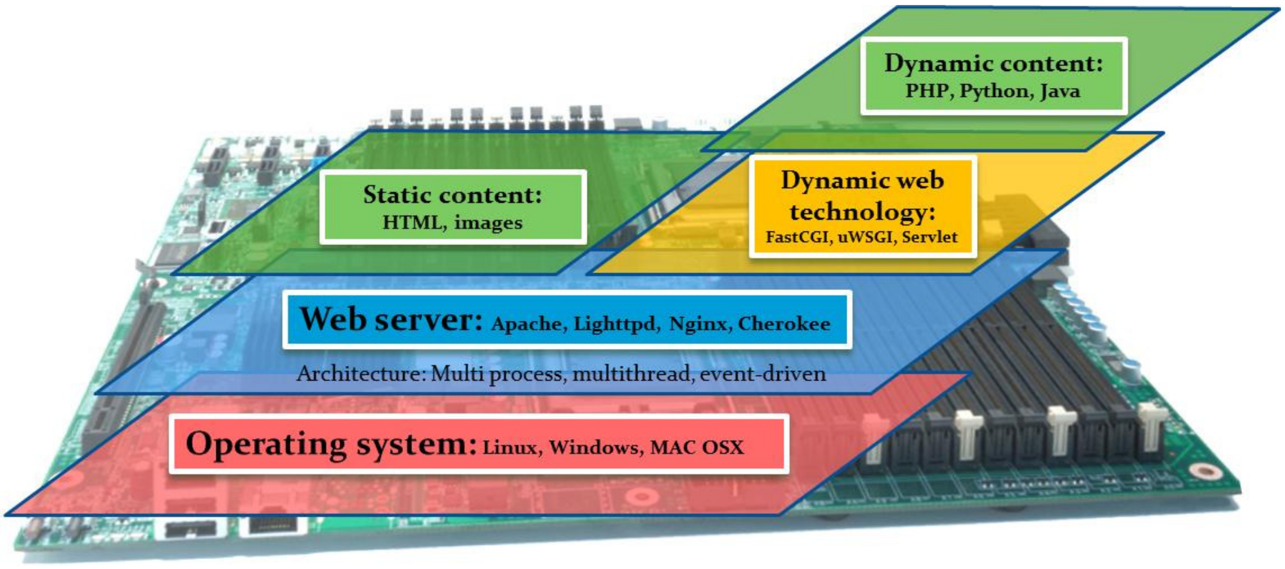
Dynamic content: PHP, Python, Java
Static content: HTML, images
Dynamic web technology: FastCGI, uWSGI, Servlet
Web server: Apache, Lighttpd, Nginx, Cherokee
Architecture: Multi process, multithread, event-driven
Operating system: Linux, Windows, MAC OSX

Fig. 2.1: Project's software stack

First of all is the hardware (meaning CPU, RAM, hard disk, etc.) where all the software will be run (see **section 3.1**). Next, is the operating system layer, where the main kernel modules as well as libraries are loaded (see **section 3.1 and 3.3**). In third place is the web server application (see **section 2.4**), which will be in charge of attend any HTTP request that is being made to the system (e.g., Apache, Lighttpd, Nginx, etc.). Following, is the language layer. Static layer can be run without any other software, as any web server understands about HTML or images. Regarding the type of language that is being used it is needed to include some dynamic web content technology (like CGI, FastCGI, uWSGI, etc.) to run it (see **section 2.2**). Finally, those dynamic content languages are run above the different dynamic web content technologies.

CHAPTER 3. TESTBED FOR WEB SERVER BENCHMARKING

After selecting the web servers that would be analyzed during this report, it is needed to establish a testbed in order to benchmark them. Is important that those benchmarks cover as many areas as possible, so we have a wide vision of which one of them perform best in some of the tests.

In order to see their performance, it is needed to maximize the possibilities of each one of the web servers, trying to execute each test case within the different configuration possibilities. This chapter will present the whole configuration of the system, including the used machines during the tests, the network between the servers, the different versions of the used software and finally, the different test cases to execute.

3.1. Used machines and software

The evolution of computer hardware in the past few years has been very high. The goal of this report is not showing the best performing web server on a piece of hardware, rather than showing a performance comparison. During the tests, different machines are going to be used, in order to simulate a network of clients and web server. Some of the machines will act as web servers, meanwhile others as clients. In next sections of this chapter is explained the difference between the distinct agents.

From the software point of view, it is needed to have the latest version of the web servers that are going to be tested. So, it is ensured to have the latest changes to the releases as well as solved previous bugs. As stable releases are not delivered monthly, it will be install the latest releases as February 2011.

Table 3.1: Main hardware and software characteristics

	PC1	**TAS01**	**TAS02**	**TAS03**	**TAS04**
Processor type	Intel Pentium D 820 (Rev. B0) @ 2.80 GHz	Intel Pentium D 950 (Rev. C1) @ 3.40 GHz	Intel Xeon 5110 (Rev. B1) @ 1.60 GHz	Core 2 Duo 6320 @ 1.86 GHz	Intel Pentium 4 521 HT @ 2.80 GHZ
Processor Cores	2	2	2	2	1 (but, two virtual cores)
RAM memory	3 GB	4 GB	4 GB	3.5 GB	3 GB
Operating system	Microsoft Windows 7 (x86)	Red Hat Enterprise Edition 5.3 (x86_64)	Red Hat Enterprise Edition 5.5 (x86_64)	Red Hat Enterprise Edition 5.5 (x86_64)	Red Hat Enterprise Edition 5.3 (x86_64)

During the course of this project are used up to five different computers, some of them focused in specific tasks (see **Table 3.1**). PC1 is a desktop Windows machine used to program all the executed scripts as well as process all the results. TAS01 is the main server, and is the place where is installed all the web servers and dynamic content technologies. TAS02, TAS03, and TAS04 are machines used to simulate a network of clients to perform the selected tests. Also, TAS02 and TAS03 could perform as a web server too, due to the fact that their hardware is not as outdated as TAS04. All the servers run 64 bit version of Red Hat Enterprise Linux, which will affect positively on the overall performance (it is necessary to take profit of more than 3 GB of RAM memory). See **Appendix 1.1** for more complete information about used hardware and software.

3.2. Network benchmark

An important factor of the scenario is to have proper network connectivity between all the clients and server machines, in order to get a good performance in all benchmarks. For this reason, all the servers will be equipped with a gigabit network interface card, to provide the best bandwidth.

Gigabit networks are not new in the market. They have been installed in computers for many years, but the cost of the equipment that are ready to process such high throughputs are higher. For workstations connected to the corporative network is enough with fast Ethernet connectivity, but when talking about servers is better to have higher speeds, like gigabit network. Another problem is that although the machines are connected through a gigabit network, it could be possible that the actual available bandwidth is lower. For this reason and before starting any benchmark, it is tested the network between both ends, using the program Iperf **[26]**.

After conducting some network tests (see **Appendix A.1**), it is possible to say that the network works in a gigabit mode, allowing us to maximize the network testing and performance. Having a gigabit network (very common in most of the cases) will become the bottleneck in cases of high-sized file transfers, where is needed a high bandwidth. In other cases, the CPU or the RAM of the machines will be the bottleneck of the system.

3.3. Software configuration

Once defined the main hardware and software specifications that are going to be used, and also the web servers that will be tested during this report, it is needed to show some operating system parameters. This gives the opportunity to test all the web servers in the same OS conditions, limiting web server’s performance variations in the way each web server is built.

3.3.1. Operating system

During this project, it is used Linux operating systems to perform the required benchmarks. As seen in previous sections of this chapter, the configuration of the different involving machines is practically the same, using also very similar Red Hat's operating system versions. In **Appendix A.1**, there are few of the main parameters of the TCP and IP configuration of each server. It is important to realize that those values are almost the same of each machine, so there is not any special tuning of it. Those parameters and values are found in /**proc**/***sys/net/core/*** and /**proc**/***sys/net/ipv4/***.

3.3.2. Web server configuration

Each web server has its own configuration file, and using different architectures. For this reason it, is needed to have as much as possible in common between them, so it can be explained why the differences in performance come from. So, each one of the tested web servers has disabled any caching, as well as compression when delivering web pages. Although running different, some components like CGI, Fast CGI or uWSGI are configured to work in the same way, configuring the spawn of processes and requests that they can handle. Each configuration file is attached in the **Appendix A.1** of this report.

3.4. Test cases

Once specified the hardware, software and the network that is going to be used, it is time to define which will be the tests to perform. It has been chosen different test cases in order to show the strengths and weaknesses of each server, and this will result in a comparison between all of them. Finally, it will be possible to know which one of the evaluated web servers best suits each type of test.

3.4.1. Definition

It is needed to establish some parameters in common to all of the tests and servers. It is going to be tested different servers, that have been programmed following different ways and using different technologies. For this reason, all tests were conducted under the following conditions:

- All tests were run multiple times to assure repeatability.
- Performance was measured in the Web Server side (to know the CPU load, and RAM usage) and in the client side (to know the requests per second and the system's load).
- During the test, no other applications were running and using resources on the system under test.

- If something is changed or added to web server's configuration file, it will be explained during the test case.

Each test will be performed in different conditions, beginning with a low load and increasing it progressively to stress out the web server. **Table 3.2** shows the different combinations that are going to be performed.

Table 3.2: Test load conditions.

		Concurrency (clients)					
Requests	1.000	1	10	100	250	500	1000
	10.000						
	100.000						
	500.000						

3.4.2. Benchmarking tools

There are different benchmark tools in the market to execute performance tests for web servers. Regarding their usage and the programing language that they are written in, the execution will differ.

3.4.2.1. Apache Benchmark (ab)

Apache Benchmark **[27]** tool is provided by the Apache foundation. It is a very powerful tool, letting the user configure the number of requests to perform and number of concurrent clients. Also, it allows saving the results of the tests in a gnuplot-file, as well as setting some POST information in the requests. Its working principle is very easy, there is no need to install the software as it comes with the Apache httpd installation (currently is installed with almost all Linux distributions).

3.4.2.2. Tsung

Tsung **[28]** is a benchmark tool written in Erlang and can be distributed to use more than one machine to perform the test. Erlang is a programming language designed for building highly parallel, distributed, fault-tolerant systems. It has been used commercially for many years to build massive fault-tolerant systems, which run for years with minimal failures. Erlang combines ideas from the world of functional programming with techniques for building fault-tolerant systems, to make a powerful language for building the massively parallel-networked applications of the future.

3.4.3. Static tests

The static test consists in generate request to static content, like an HTML only web page or an image. This test will show the performance of the web server in hits per second. To perform the test it will be started the web server in the TAS01 machine. After realize that the server is up and running, it will be run the different static tests that are prepared. Previously to the tests, it is loaded the web server with all the needed files, as for example the HTML file and some images of different size. Files are relatively small compared to the bandwidth of the network; but the fact that it is going to put the server into stress, will mean that the number of requests per second generated will be high enough to fill up the whole gigabit network.

3.4.3.1. ST-1 HTML

This benchmark consists in request an HTML file of 168 bytes. This file only includes few lines of code, where is only shown a test sentence, *This a webserver test page*. Here it will be expected the network not to be the main problem, rather than the capacity of the web server of getting profit of the processor or getting the processor extremely loaded.

3.4.3.2. ST-2 Image small

This benchmark consists in request a small image file of 7,500 bytes with a dimension of 203x61 pixels. This image is a small one, but is about 40 times larger than the previous test. Here it will be expected the network to be a possible bottleneck, although running in gigabit mode.

3.4.3.3. ST-3 Image large

This benchmark consists in request a larger file with a size of 83,572 bytes with a dimension of 1600x1200 pixels. This image is 10 times larger than the previous one. Here it is expected the network to be the main problem, although running in gigabit mode.

3.4.4. Dynamic tests

The dynamic test consists in requesting pages that need to be load dynamically form the server. These pages are written in dynamic languages such as PHP, Python, Perl, etc. There are different ways to execute those languages regarding the web server technology that it is chosen. So, this test will show the differences between running the dynamic language file in a module inside the web server in front of CGI or FastCGI, Servlet or JSP. During tests three common dynamic languages are used in web pages like PHP, Python and Java. They are chosen because their use in the web environment is extended, so it is important to know their performance.

3.4.4.1. DT-1 PHP

This benchmark consists in requesting a dynamic page written in PHP. The page will be a PHP file of 14 bytes, only returning a *Hello, World!* sentence. As there are different types of web servers, three different types of tests are defined according to the possibilities of each of them. Here there is no need to calculate the maximum number of requests that the network can handle, due to the fact that processing a dynamic language will load the CPU of the machine, limiting the requests per second. Although being a very small file, it will depend on how it is executed in the web server side (module/CGI/Fast CGI) to get more performance.

- *DT-1-1 MOD_PHP*

This test can only be done in Apache due to the fact that is the only one that will have the PHP interpreter loaded as a module in the same web server. So, it can be expected that this will give an advantage to Apache in this test, getting better performance numbers in the whole test.

- *DT-1-2 CGI*

This test can be done in almost all of the tested web servers. CGI protocol is enabled in every one of the installed servers, with the exception of only Nginx, which does not recommend executing and relays on third party extensions. As said before, executing in CGI mode will result in the creation of a new process each time that a file accomplishes the rules specified in CGI configuration. There is no process spawn, so each time that those files are requested, a new process is started increasing the load of the web server.

- *DT-1-3 FCGI*

Fast CGI is enabled in almost all of the installed web servers, but the difference relays in how the web server call the Fast CGI protocol to spawn new processes. Comparing with CGI, it is expected to increase dramatically the performance of the web server, due to the fact that it will be configured how many PHP Fast CGI processes to spawn and how many requests each process will handle. As the processes will not be killed when finishing their job, this will reduce the cost of creation and destruction of processes in the operating system.

3.4.4.2. DT-2 PYTHON

This benchmark consists in requesting a dynamic page written in Python. The page will be a Python file of bytes 750 bytes. As there are different types of web servers, we define three different types of tests according to the possibilities of each one of them. If it is calculated the maximum number of requests per second, it will be a high number, but it will not be reached. As there is the need to interpret the language, the performance of the web server will decrease.

- *DT-2-1 CGI*

This test will consist in running the python script in CGI mode. As said in the PHP case, it is expected that the performance of the system running in CGI mode will be lower than for other modes. Not all of the installed web servers support CGI protocol, as Nginx comes without support for it. For the others, it will be tested to compare the obtained results within them.

- *DT-2-2 FCGI*

This test can be done in all of the tested web servers. As said with the previous dynamic tests, if it is compared with CGI it is expected an increase of the web servers' performance. For Fast CGI test, it is needed to change the script in order to include the needed modules to run it. It is going to be used the *flup* library for python, which will let to execute the script in Fast CGI mode. The structure of the script changes a little bit, just to include the needed libraries and also to define the application to execute.

- *DT-2-3 uWSGI*

Not all web servers spawn Fast CGI processes in the same way, driving to some execution problems of the scripts. For those in which Fast CGI is not running in the best possible way, there is another option called uWSGI. It is expected the performance of the web server match the obtained for Fast CGI, or even higher.

3.4.4.3. DT-3 SERVLET

This test consists in requesting a Servlet page to the web server. Not all the tested web servers support delivering Servlet pages, so it can only be tested in Tomcat web server. A Servlet extends the Java language capabilities into the request-response programming model. Tomcat offers the possibility of running Servlets in the core of the web server. Here it will be benchmarked the Tomcat Servlet feature.

3.4.4.4. DT-4 JSP

This test consists in requesting a JSP page to the web server. JSP test, as well as Servlet test, will be performed only in Apache Tomcat. Tomcat offers JSP execution possibility. It is expected that the performance of this test will be higher to the Servlet one.

3.4.5. Keep-alive tests

Each of the previous tests, being static or dynamic, will be performed in twice: with and without keep-alive. Since the introduction of keep-alive function in the

HTTP 1.1 standard, the performance of the web server is increased, as one connection could make more than one request. For this reason, this test is an important one, as will show how the performance changes by switching this functionality. It is expected that using keep-alive during the tests will put more load in the web server, but increase the performance of it.

3.4.6. HTTPS tests

Until now all tests are perform with HTTP, the default protocol to request a web page in the Internet. But nowadays, is important to secure connections, as more confidential information is shared between insecure networks. Securing connections between the user and the web server will have an impact in the performance of the web server. It will be useful to describe this decrease of the performance and if it is considerable.

It will be tested some static and dynamic content in HTTPS mode to see how it performs against not secure protocol. The requested files will be the same as in previous cases:

- *HTTPST-1 HTML*
- *HTTPST-2 PHP*
- *HTTPST-3 Python*

To be able to perform those tests it will be configured the web servers to support secure connections. For this reason it will be generated some certificates, in order to enable the SSL feature in the web server. For generating all the certificates the OpenSSL (**Ref. [28]**) application is going to be used. In the Appendix are the followed steps in order to create the needed certificates.

3.4.7. Load test

Until now it has been tested the performance in requests per second of the web server. The load test goes beyond in the performance tests by setting the creation of new clients; those will request some of the web pages during some defined time. Finally, this will give an idea of the capacity of the web server, as it will tell if it can handle the workload that it is being tested. Here it will be defined how many clients will be created each second, as well as how many requests will each client do while is running, and the duration of them. Also, the workload could be distributed between different machines, giving the opportunity to increase even more the load of the web server. It is possible to simulate such test by using one of the previous web pages, HTML, PHP and Python; instead it is chosen to install a blog web page.

Blogs are being used by a lot of people around the world to show or comment their experiences. Those experiences do not share any in common, as some of them talk about trips, sports, food, or even technological reviews. In order to perform such tests, it will be installed a Wordpress blog site. Wordpress (**Ref. [29]**) is being used by thousands of people to share its experiences, so it could be a very good point to show the real performance of the web server in a real workload.

In order to make the test case more realistic, some dynamic language accelerators as well as some caching mechanism will be installed. Those plugins will be common for all the installations, so all the web servers could benefit from this boosts of performance. As many frameworks found on Internet, they use PHP as a main dynamic language as well as HTML, MySQL database and images. To improve PHP performance, PHP accelerators will cache some of the used instructions. Disk and RAM cache will accelerate the serving time of the rest of the web page. Database access could also be a bottleneck, as latency for seeking and adding content is slow.

To perform the test it will be set the creation of new clients on three different phases, with a total duration of one-hour time frame. Each client will make two requests to the web server, one to the main page and the other to a defined post. Tsung tool will provide statistics and also some graphics about the performance of the web server. The configuration file of Tsung is based on XML (in **Appendix A.2.4** is an example used in this test). Ganglia (**Ref. [30]**), which is a distributed monitoring system, will report statistics about CPU and RAM usage.

CHAPTER 4. TEST RESULTS

Once all the web servers are selected, and also all the different tests cases are setup in the different machines, it is proceed with the obtained results. This chapter focuses in presenting the different performance charts, regarding the tests and the web servers.

Not all the test cases are equal, and some of them require more results in order to see how the different web servers scale when changing some configuration parameters. It is important to show the performance gaps and try to explain whether they appear or not. Moreover, all the tests are made with and without enabling keep-alive; in order to show the differences in performance when sharing established connections. Although tests are performed with different concurrency levels and number of requests, it will only be shown the results of the 100,000 request with all the concurrencies. Performance with fewer requests have high variability, meanwhile higher counts do not offer any performance improvement.

It will be followed the same pattern that was used to describe all the tests cases. In first place, it will appear the results of static contents, then the results of dynamic contents, next the results regarding SSL test cases, and finally, the load test.

4.1. Static tests

4.1.1. ST-1 HTML test

The first static test consists in return a 168 bytes HTML only web page, where it is shown the phrase: “This is a webserver test page”.

Before starting with the charts, it is important to highlight three well different results that were obtained, being the first one about do not enabling keep-alive. Using HTTP 1.0 protocol affects the overall performance, as it lowers the server CPU load but, is limited by how speedier the server opens new connections. Afterwards, it was enabled keep-alive functionality, using HTTP 1.1 protocol, which increased the CPU load in order to maintain connections opened, but also increased the performance. The importance of this test is to appreciate the performance variation between both cases, and the relative performance difference within web servers.

The third result that is important to mention in this test case, is the increase of the performance when tuning event-driven web servers (e.g., Lighttpd or Nginx). The default configuration for those servers is only one process with a single thread; but, it can be increased to take advantage of the multi process architecture of the server. It will be shown different charts varying the number of processes, in order to find which the best-suited configuration is. Finally, it will be compared the best performer web servers of this section in order to get the best web server for this test.

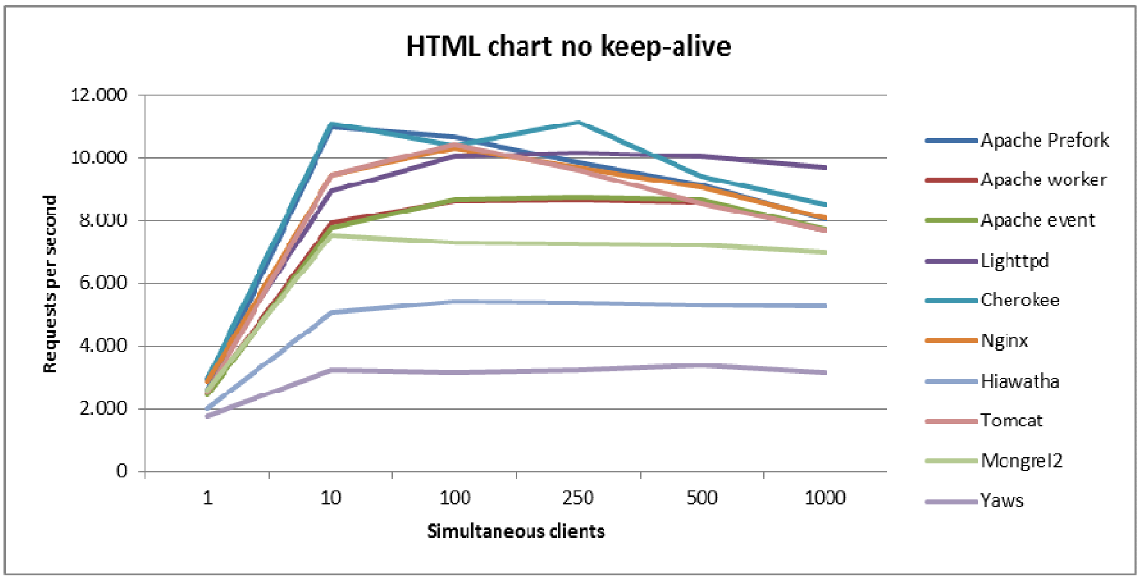

Fig. 4.1: HTML chart no keep-alive test results

Fig. 4.1 shows the HTML chart between the different tested web servers. In this case, average performance is between 8,000 and 11,000 requests per second, only Hiawatha and Yaws being far behind the others. Apache Worker (multi process multithread) and Event (event-based) configurations perform in the same way. Only Apache's Prefork (multi process single thread) configuration performs better, about 3,000 requests per second more, in the best case. When reaching 500 simultaneous clients the performance of the three Apache configurations is similar, about 8,000 requests per second.

Lighttpd and Nginx web servers have a very similar performance, both reaching 10,000 requests per second with 100 clients at the same time. Performance difference starts to appear at 500 simultaneous clients, Nginx performance decreases quicker than Lighttpd. In this case, both Lighttpd and Nginx were tested with only one process running. On the other hand, multithread web servers such as Cherokee and Tomcat have a similar performance behavior, having Cherokee slightly superior performance in front of Tomcat. Both have the same performance decrease when reaching above 250 simultaneous clients, and with 1,000 clients the performance is almost the same, around 8,000 requests per second.

Mongrel2 gets better performance than Hiawatha, which is a very good notice as it is a newer web server using a very different architecture approach. Yaws gets the worst performance rate, less than 4,000 requests per second. This can be caused by the fact that without keep-alive, Yaws only uses one processor, which limits its performance.

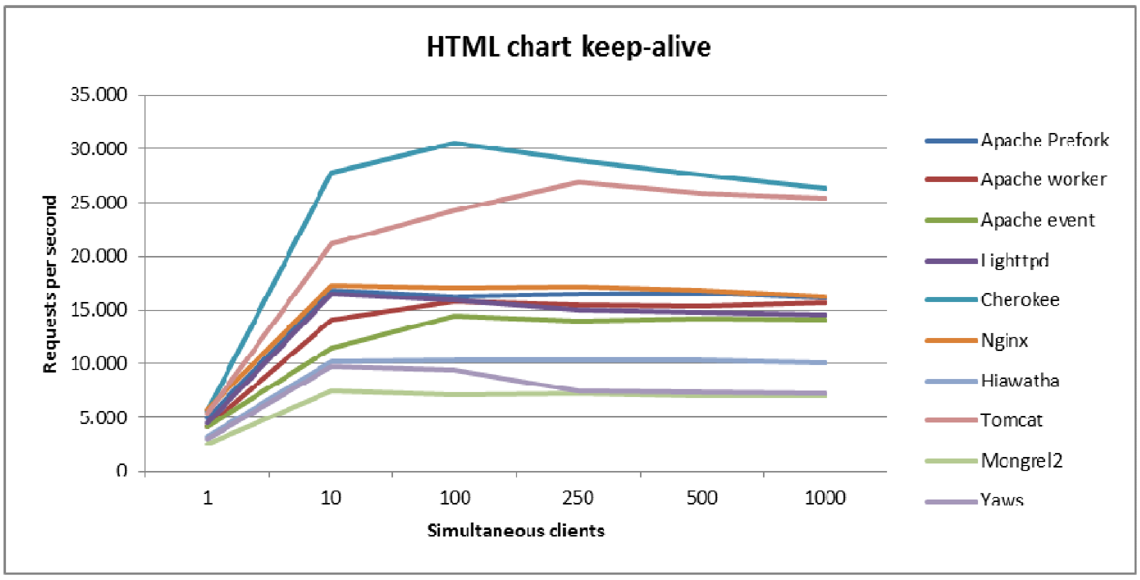

Fig. 4.2: HTML chart keep-alive test results

Enabling keep-alive feature, HTTP 1.1 protocol, increases the performance of the system, as it shares connections between requests. **Fig. 4.2** shows the result increase that, in some cases, reaches up to 30,000 requests per second. Apache has a very similar performance for the three different configurations. Only at the beginning of the test, between 1 and 100 simultaneous clients, performance difference is around 5,000 requests per second. When reaching 100 simultaneous clients, the results of the three configurations are around 15,000 requests per second. The performance increase by using keep-alive for Apache against not using it is more than 5,000 requests per second, which represents an increase of more than 50%.

Event–driven web servers got almost the same result, reaching 17,000 requests per second for Nginx and 16,500 for Lighttpd, in the best case. Nginx seemed to perform more stable when increasing simultaneous clients; while Lighttpd decreases its performance near 15,000 requests per second. Speaking about multithread web servers, Cherokee and Tomcat, both obtained performance around 30,000 and 27,000 requests per second, respectively. Cherokee result increased almost 270% by enabling keep-alive feature, which is a large increase. It is possible to see that Tomcat reaches its maximum performance range slowly than Cherokee, but maintains it as simultaneous clients increase.

Hiawatha only reaches 10,000 requests per second, although it doubles its previous result, but it is still 50% slower than Apache, Lighttpd and Nginx. Mongrel 2 and Yaws had a very low performance, which is not a good outlook for them. On one hand, Yaws uses the two available processors, but falls far ahead from best performing web servers. On the other hand, Mongrel2 does not take profit of the whole CPU capabilities, which leaves it as the worst performing web server of this test.

Event-driven based web servers can also be configured to take advantage of multi process systems by enabling the ability to fork its main process. As the aim of the test is to show the best performing web server for this scenario, it is configured Lighttpd and Nginx with different number of active processes. Taking into account previous results, performance without keep-alive feature enabled is almost the same (see **Appendix A.2.1**) for any configuration. The only difference is that with more than one Lighttpd worker, the top performance is reached earlier (11,000 requests per second in the best case), but performance is not as stable as before when increasing simultaneous clients.

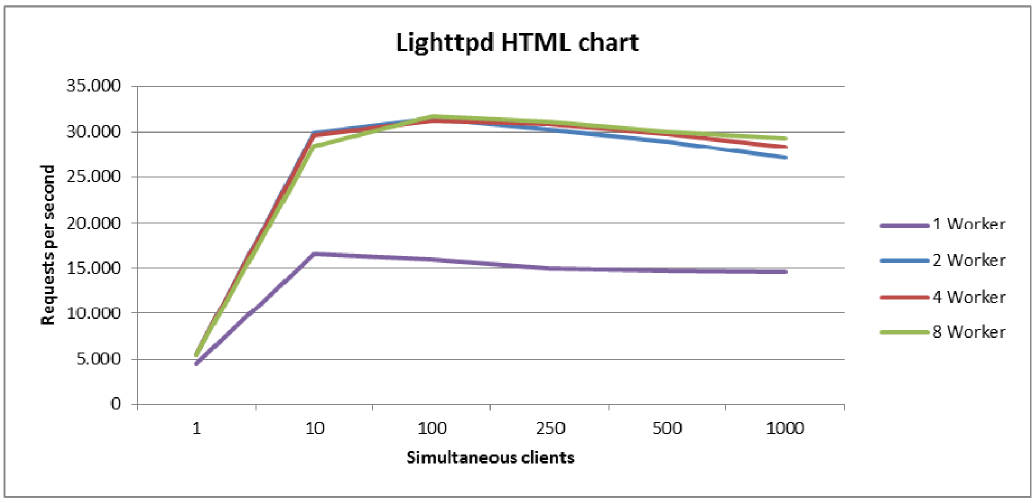

Fig. 4.3: Lighttpd HTML keep-alive chart

But, when enabling keep-alive we get very different results. **Fig. 4.3** shows the performance obtained during this test. It is possible to see a big increase when configuring Lighttpd to fork in more than one process. Performance increase is about 100%, increasing from 15,000 requests per second with one process up to 30,000 requests per second with more than one process. Lighttpd does not recommend forking as it could break some of the kernel modules of the web server, but for increasing the performance it is advisable. Also, they recommend using twice the processes as the number of processors that the machine has, in this case it would be 4 processes because the machine is a multicore system.

The other event-driven web server is Nginx. The results were expected to be similar to the previous one, Lighttpd. No keep-alive HTML test, results in a top performance of 10,000 requests per second for any configuration (see **Appendix A.2.1**). As in the case of Lighttpd, there is no much difference between configurations as they behave very similar.

Fig. 4.4 shows the scalability of Nginx in different worker situations enabling keep-alive feature. It is possible to see that configuring the web server by default, with only one process, has a measurable impact on the performance. Meanwhile, using more than one process increases performance up to 28,000 requests per second, in the best case. Forking the main process to more than 2 workers results in a very similar performance, so there is no special performance gain when using even more processes. Nginx official documentation recommends forking the main process to be equal than the number of available processors of the system. In this case, as it is shown in **Fig. 4.4**, the best result occurs when having 2 Nginx workers at the same time.

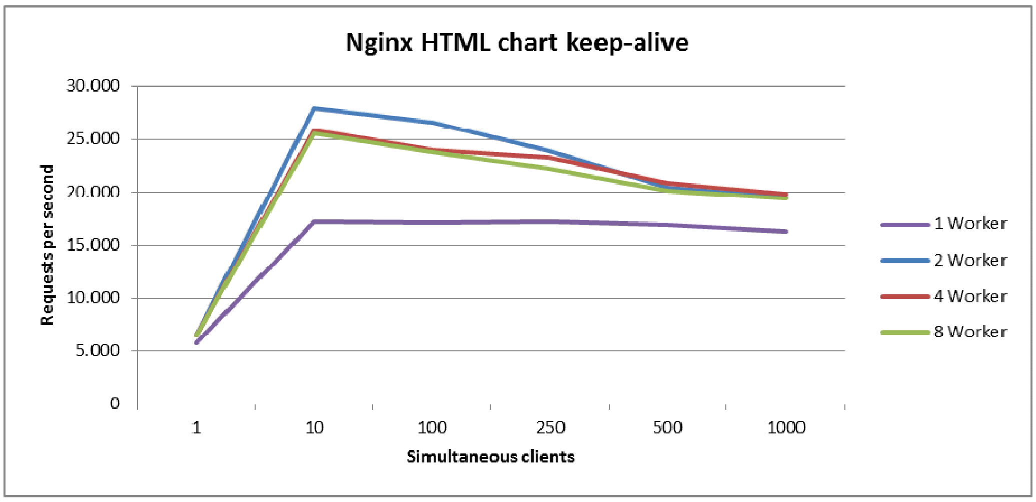

Fig. 4.4: Nginx HTML keep-alive chart

4.1.2. ST-2 Image small test

This test case consists in requesting an image, which is 7,500 Bytes long. It is not a large file and, as calculated in the previous chapter, the top performance is expected to be up to 15,666 requests per second. The procedure is going to be the same as with HTML test case, so the followed schema of the section will be the same as in the previous one. First, it will be commented the evolution of the usage or not of keep-alive functionality and, finally, the event-driven scalability.

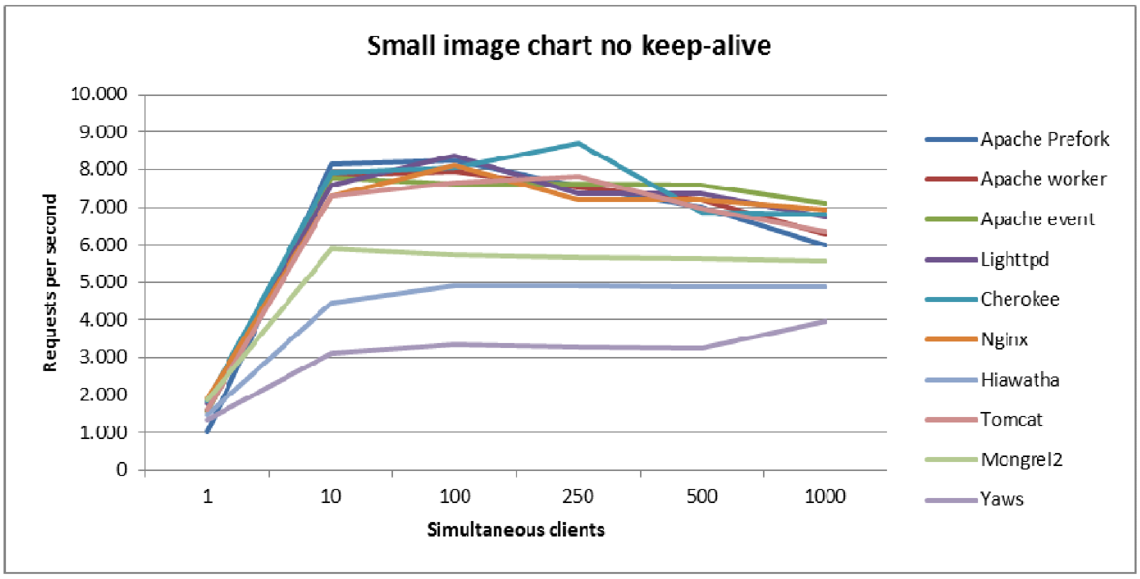

Fig. 4.5: Small image no keep-alive chart

Fig. 4.5 shows the results of the small image test case. Performance of almost all web servers is equal, being up to 8,000 requests per second and slowly decreasing when raising the number of simultaneous clients in the system. There is not much difference between choosing a multi-process, multithread or event-driven web server, as they all get the same performance. But, it can be seen that there are three web servers with lower results. Mongrel2 overtakes Hiawatha and Yaws, which is surprising given the previous test results. Hiawatha shows another time its performance limits, only reaching up to 5,000 requests per second. Yaws has the same problem as in the previous test, only using a single processor, which limits its performance in no keep-alive tests.

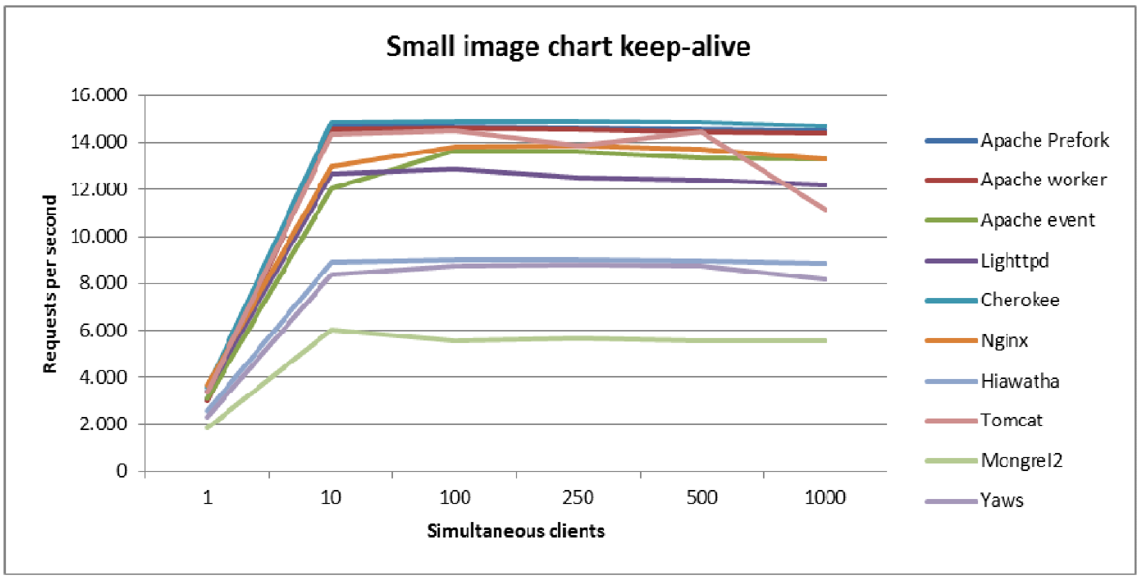

Fig. 4.6: Small image keep-alive chart

When using keep-alive of HTTP 1.1 protocol, performance of the web servers increase. **Fig. 4.6** shows the results of this test. Performance is between 12,500 and 15,000 requests per second. Apache Prefork and Worker results are very similar, only slightly lower for the Event configuration. Lighttpd's performance is slower than the other high performance web servers, being near 13,000 requests per second. The other event-driven web server, Nginx, tops in 14,000 requests per second, which is a very good result, near the multithread Cherokee and Tomcat.

Hiawatha and Yaws get very close performance results, up to 9,000 requests per second, which is a gain of 80% and 200%, respectively, but is still a 55% slower than Cherokee or Apache. Mongrel2 performance is limited by its architecture, only getting 6,000 requests per second, being the same as in the previous case.

Now, it is turn to see the performance increase when using event-driven web server. It is not possible to expect such a high increase of performance as happened with HTML test case, because the theoretical maximum rate is 15,666 requests per second. Nevertheless, it is expected some increase of the performance. Without keep-alive, Lighttpd performance does not increase a lot when configuring the web server with more than one process (see **Appendix A.2.2**). The best performing configuration is when there are 8 simultaneous workers instead of the Lighttpd's recommendation, which is 4. But, the performance gap between both configurations is small, less than 1,000 requests per second (15% approximately).

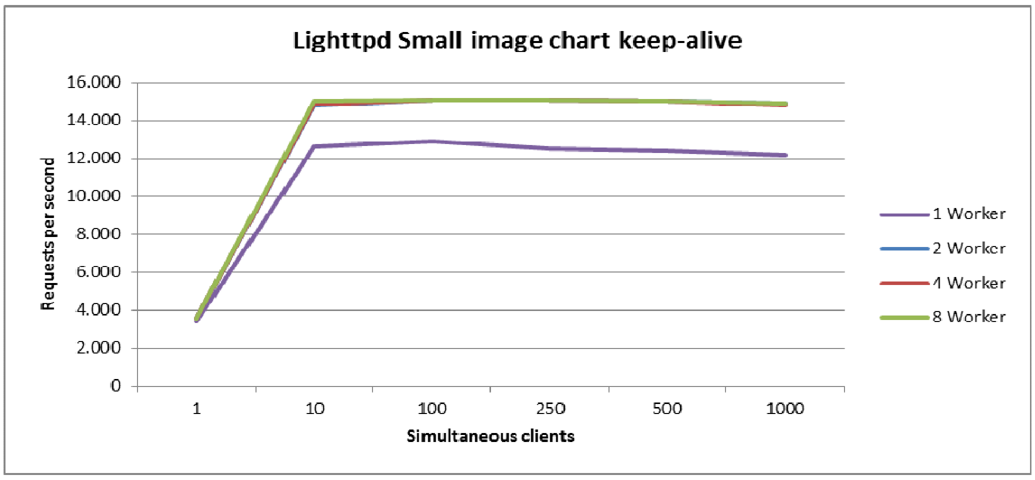

Fig. 4.7: Lighttpd small image keep-alive chart

With keep-alive enabled it is possible to see that performance increases significantly. **Fig. 4.7** shows this behavior between single process and multi-process configuration. Difference between the configurations is about 2,500 requests per second. The top performance rate is around 15,000 requests per second, near the theoretical maximum rate.

Nginx behavior in this test is expected to be in line with results obtained for Lighttpd, as both share the same architecture. Not using keep-alive gets almost the same result, between 7,000 and 8,000 requests per second for each configuration (see **Appendix A.2.2**). Nginx recommendation is to use 2 active worker processes, and in this case does not matter how many they are, as the performance is almost the same in all cases.

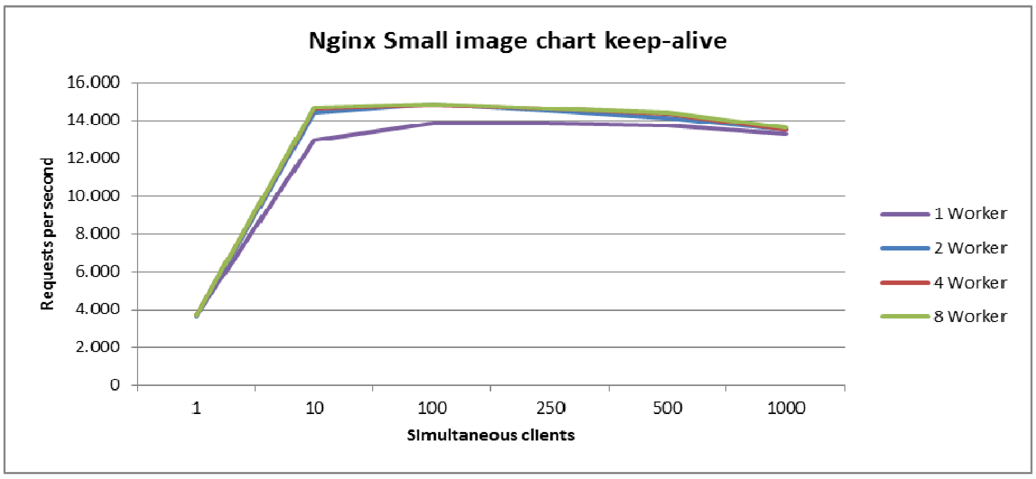

Fig. 4.8: Nginx small image keep-alive chart

Nginx performance is very similar to Lighttpd, as top performance is limited by the network bandwidth. **Fig. 4.8** shows the configuration differences between the four formations in keep-alive mode. Performance using only one worker is slightly lower than multi-process configuration, of 1,000 requests per second. In any case, the performance of the web server is being limited by the network bandwidth.

4.1.3. ST-3 Image large test

The third static test consists in requesting a large image, which sizes up to 83 KB. Performance of this test is limited by the fact that a big file will fill in the whole network bandwidth, getting lower rates. As calculated in the previous chapter, the maximum expected performance would be around 1,400 requests per second.

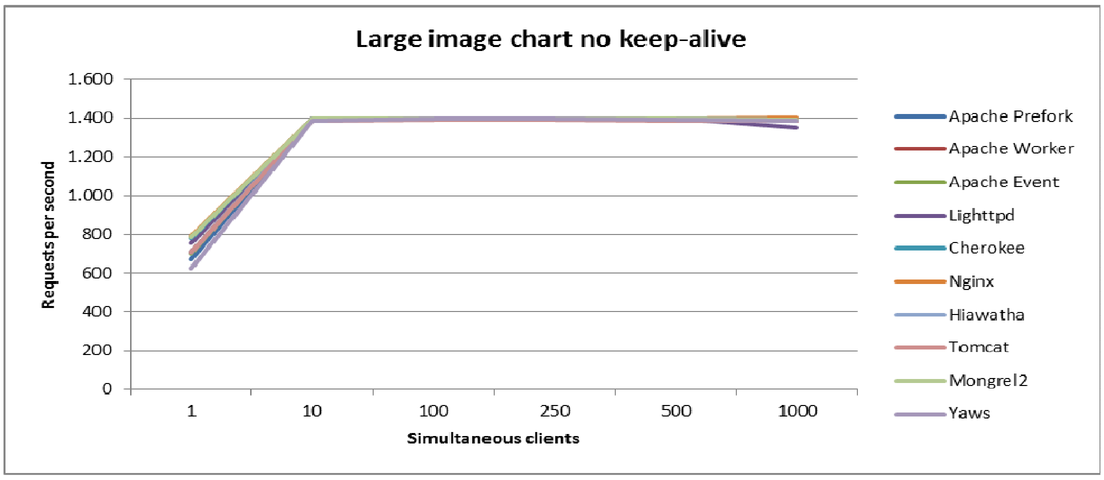

Fig. 4.9: Large image no keep-alive chart

The first chart to be displayed is the performance results without keep-alive enabled. **Fig. 4.9** shows that performance for each one of the tested web servers in this report. It is possible to see that all of them show the same results, with a top rate around 1,400 requests per second, which is the maximum we can expect with a gigabit network card.

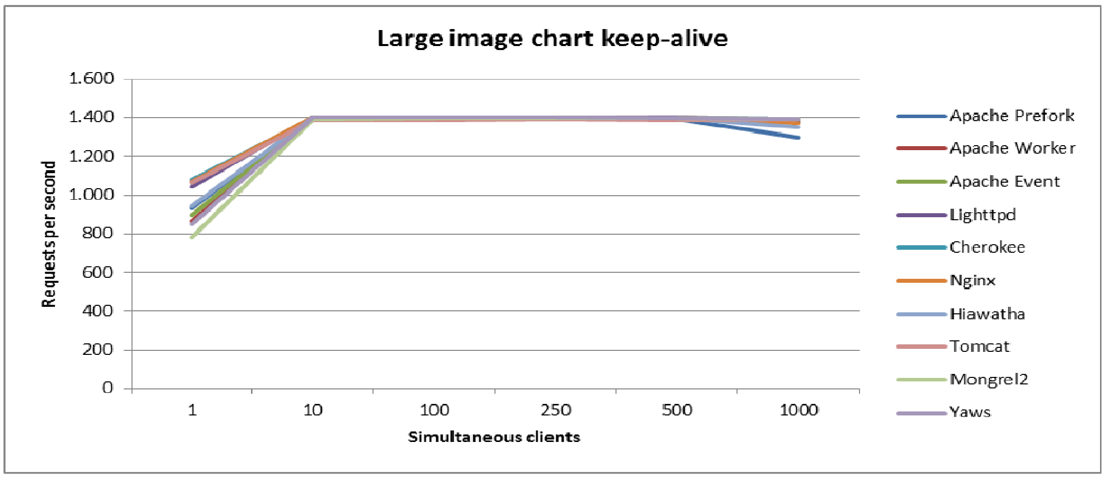

Fig. 4.10: Large image keep-alive chart

Fig. 4.10 shows the large image result chart with keep-alive enabled. In this case, results are very similar to the previous one, with the same performance, 1,400 requests per second. Again, his performance is limited by the bandwidth of the network, so the web server is not fully stressed. So, there is not an increase for event-driven web servers, because the server software is not the limit in this case.

4.1.4. Static tests conclusions

Taking into account all the obtained results in this first section of the chapter, it is possible to take some conclusions based on those numbers. In case of a HTML file, the clear winner of the test case was Lighttpd, which outperformed the rest of the web servers. Cherokee web server is behind it by a narrow gap, making it also suitable for this task. Nginx is also a good option, as its performance is close to Lighttpd, although it decreases quicker than the others. Although Apache web server is the king of the web servers, it offers lower performance results in the HTML test, far away from other solutions. The three different multi-process configurations of Apache have a very similar performance, being better the prefork, which is the default installation.

Speaking about a small image, the winners are Lighttpd and Cherokee another time, but here the differences with the other web servers are small or negligible. Network bandwidth limits the performance of the web server, but having in mind the results of the HTML test, both web servers can be used in both scenarios, ensuring the best performance. The rest of the tested web servers (Apache, Tomcat, Nginx or Hiawatha) also perform as expected, generating the same performance of Lighttpd or Cherokee. But, they do not perform as well as Lighttpd or Cherokee (taking into account also HTML test case). The last static test consisted in requesting a large image, and like in the small image test, the network bandwidth is an issue. Here there is no successful candidate, as all the web servers perform exactly the same.

Summing up this section, Lighttpd and Cherokee are clear winners of the static test cases, as they perform better than other solutions. They have different architecture, as Lighttpd is an event-driven software and Cherokee is a single process multithread application. Cherokee is easier to configure than Lighttpd or any other web server, as it has a graphic user interface that can be access through a web browser. Although Lighttpd configuration is not as easy as Cherokee, it is still a comprehensive configuration file. Also, mention that official Lighty web page has a lot of interesting information. Nginx’s performance is high enough to be considered also a very good option for those tasks. Apache web server does not show its strengths during the first bunch of tests, because it is not optimized to deal with static content.

The worst performing web servers are Hiawatha, Yaws and Mongrel2. Hiawatha performance is surprisingly limited, although it shares the same architecture as Cherokee and Tomcat. Yaws is the only web server that uses Erlang in its core, which is designed to allow thousands of simultaneous users, as it was designed to work with telephonic gateways. Maybe, it would have shown its performance strength if in the tests it has been reached even more simultaneous connections. Mongrel2 which design is completely new, is limited by the usage of only one processor.

4.2. Dynamic tests

Dynamic languages are more computational expensive than static, but provides web pages with some dynamism that allows the creation of desktop-class applications for them. This set of test cases will try to show the performance issues that web servers face when dealing with such languages. Tests will involve the use of PHP, Python and Java, which are highly used languages in web creation.

4.2.1. DT-1 PHP test

The first dynamic language that is going to be tested is PHP. This language is very used in web programming, so it is an important part of the test cases. There are different ways of running PHP in the tested web servers. For instance, Apache web server can run it as a kernel module, with mod_php module, as FastCGI, or CGI. It has already been explained the difference between CGI and FastCGI in previous chapters, but here it will be shown the difference in performance.

But, not all the configurations of Apache can run PHP as a module, since PHP is not thread safe by default. For this reason, only when running Apache as Prefork mode it is going to be tested mod_php, meanwhile for Worker or Event modules it is going to be run as FastCGI. Lighttpd, Cherokee and Hiawatha can run PHP as FastCGI and CGI, but Nginx only as FastCGI, because CGI is not officially supported. Tomcat only has a CGI connector to run PHP language, although it is supposed to run Java applications.

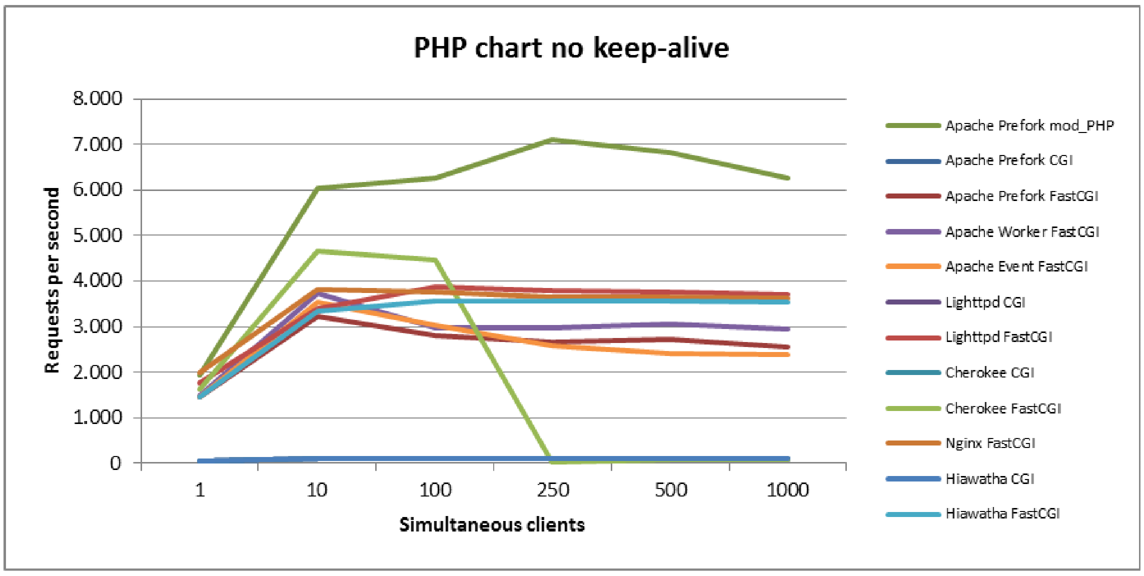

Fig. 4.11: PHP no keep-alive chart

Fig. 4.11 shows the results of all the web servers serving PHP web pages without keep-alive enabled. There are three different groups; the first one is Apache prefork with mod_php loaded, being its performance configuration between 6,000 and 7,000 requests per second, higher than any other. It is important to remember that mod_php is a loaded kernel extension into Apache, and this is the reason why the performance in this case is so high.

The other group is those web servers running FastCGI protocol, and they can also be divided in three subgroups. The first subgroup is Cherokee web server, which reaches up to 4,500 requests per second, but as the number of simultaneous clients in the system increases, the web server gets the entire CPU load, which is 200% (remember that is a dual core machine). This issue provokes that there are no FastCGI processes attending requests, so the overall performance decreases to less than 100 requests per second. The servers of the second subgroup are those based on event-driven architecture, Lighttpd and Nginx; both have very similar results, around 3,500 requests per second. It is important to remember that they are tested with only one worker configuration, and possibly there is space for scalability in the system. The last subgroup consists of Apache Prefork, Worker and Event configurations running FastCGI protocol. Performance of those three web servers is lower, getting between 2,500 and 3,000 requests per second.

The last group of web servers is those using CGI protocol. As CGI needs to create a new process each time there is a new request for a dynamic web page, in this case PHP file, its performance is very slow, near 100 requests per second. CGI protocol is being replaced by FastCGI, as it offers better security features as well as improved performance.

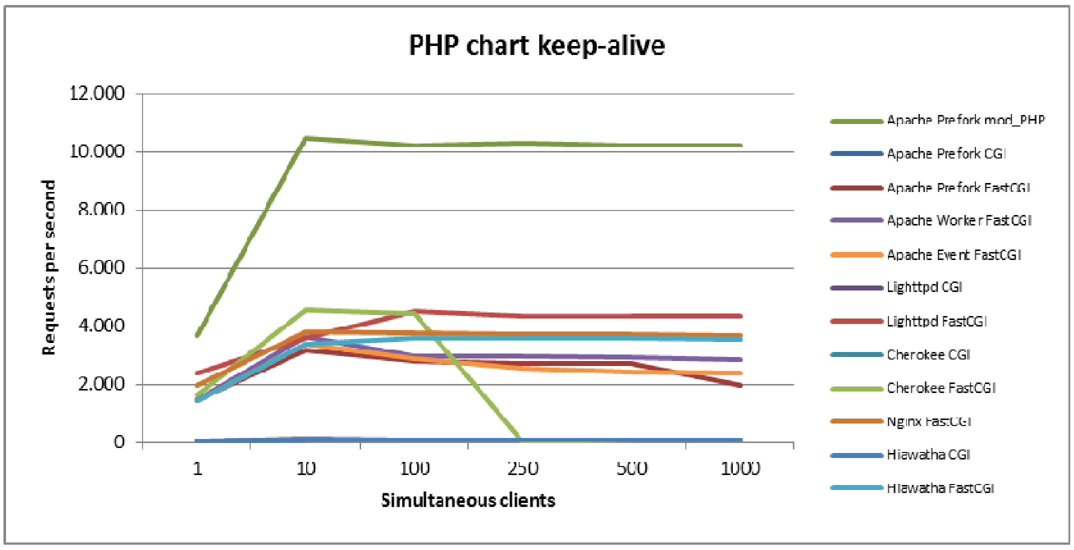

Fig. 4.12: PHP keep-alive chart

Now is the turn to see the performance of web servers serving PHP with keep-alive feature enabled. **Fig. 4.12** shows the results of each one of the web servers. Is used the same classification as before. The best performing web server in this test is Apache prefork with mod_php module loaded into the kernel, getting around 10,000 requests per second, which is a very high rate.

In the second group, being those that are running FastCGI protocol, it is possible to see the same behavior than before, with Cherokee getting a high performance until there are more than 100 simultaneous clients that start to decrease. Lighttpd, Nginx and Hiawatha get similar performance, around 4,000 requests per second. So, again, Apache configurations running FastCGI protocol got lower performance than other web servers, resulting between 2,000 and 3,000 requests per second. Finally, all those running CGI protocol got the lowest performance of the test, with barely 100 requests per second.

In static tests it has been proven that Apache web servers were not the best option in order to server static content. But, in dynamic content it is possible to see a major performance increase when running it as a kernel module against other web servers. **Fig. 4.13** shows the performance gap between running mod_php and FastCGI protocol in Apache configurations.

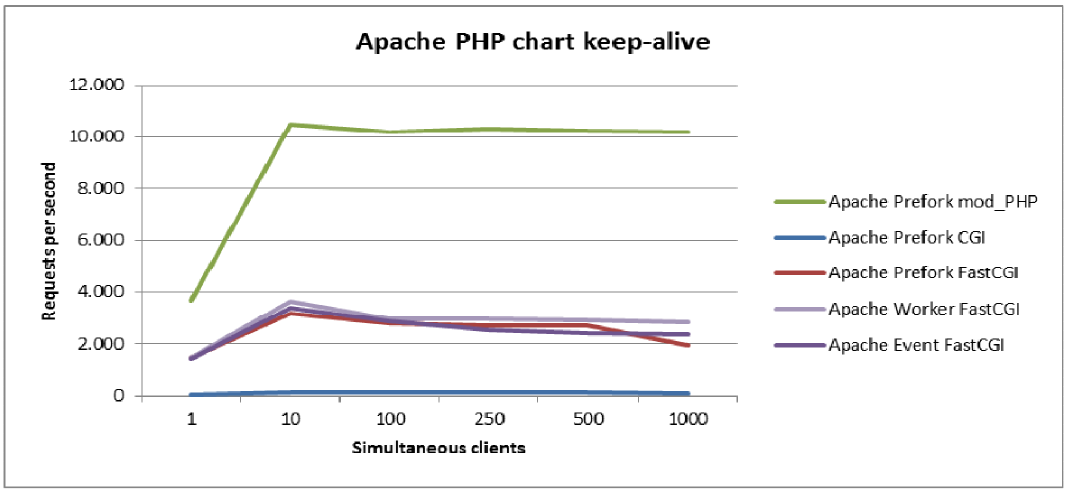

Fig. 4.13: Apache PHP keep-alive chart

Running mod_php gets a top performance around 10,000 requests per second, which is an increase of 333% regarding the use of FastCGI protocol. Also, in **Fig. 4.12** it is clear that mod_php outperformed all the other tested web servers in this report. Meanwhile, performance is the highest one of the PHP test case, resource consumption is also higher (see **Appendix A.2**). RAM process consumption increases as many Apache processes are being created. Furthermore, it is important to know the increase of performance when running event-driven web server architectures with more than one process at the same time. For this reason, different configurations are set up to be tested, similar to what was done in the HTML test case.

Enabling keep-alive feature in Lighttpd increases the performance of the system, as it is shown in **Fig. 4.14**. Configurations with more than one worker increases their performance, reaching a top result of 6,000 requests per second; although, it gets less stable as increasing the number of simultaneous clients. With only one worker configuration, performance is stable during the test, but is slower, getting 4,500 requests per second. CGI performance is the same as before, only getting 100 requests per second.

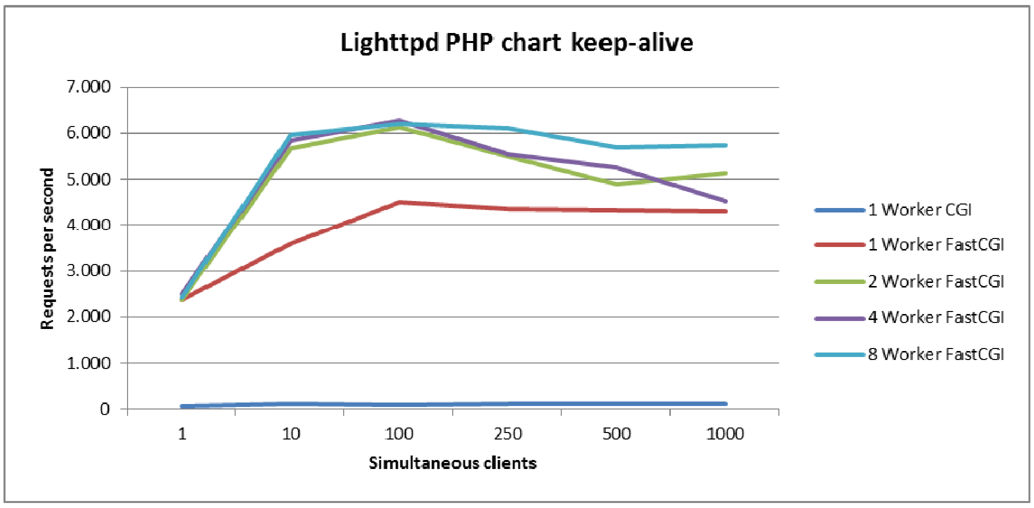

Fig. 4.14: Lighttpd PHP keep-alive chart

Performance results obtained enabling Nginx's keep-alive are almost the same, as it is shown in **Fig. 4.15**. Execution with one and two worker configuration is stable along the test, getting 3,800 and 4,200 requests per second respectively. Performance increases when using four and eight worker configuration, up to 5,100 requests per second, but it starts to decrease quickly when reaching 250 and 500 simultaneous users respectively.

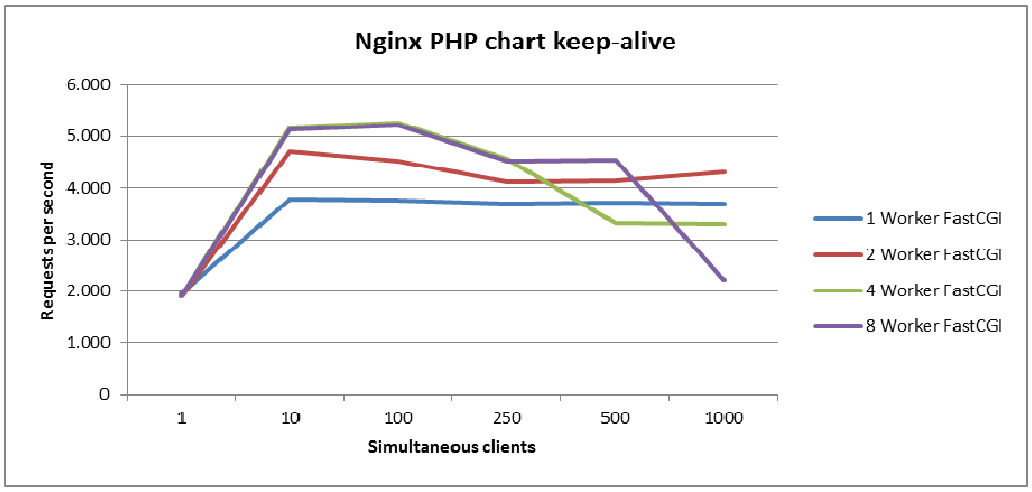

Fig. 4.15: Nginx PHP keep-alive chart

Comparing Lighttpd and Nginx, it is possible to see that Lighttpd has a better overall performance than Nginx. Even in one worker configuration, Lighttpd performs similar to two Nginx worker processes. Moreover, Lighttpd performance when increasing the number of active workers is higher, reaching up to 6,000 requests per second; while Nginx tops at 5,100 requests per second. As simultaneous clients increase, both web servers, in multi worker configuration, decrease their performance. This performance decrease is more noticeable in Nginx's case.

4.2.2. DT-2 PYTHON test

The second part of these dynamic test cases consists in testing some python scripts. As explained in **Chapter 3**, it consists in showing in the screen a string of 750 bytes, which will be executed with means of a Python interpreter. As happened with PHP tests, there are several different Python interpreters such as CGI, FastCGI, uWSGI, etc., although not all of them are supported by all of the tested web servers.

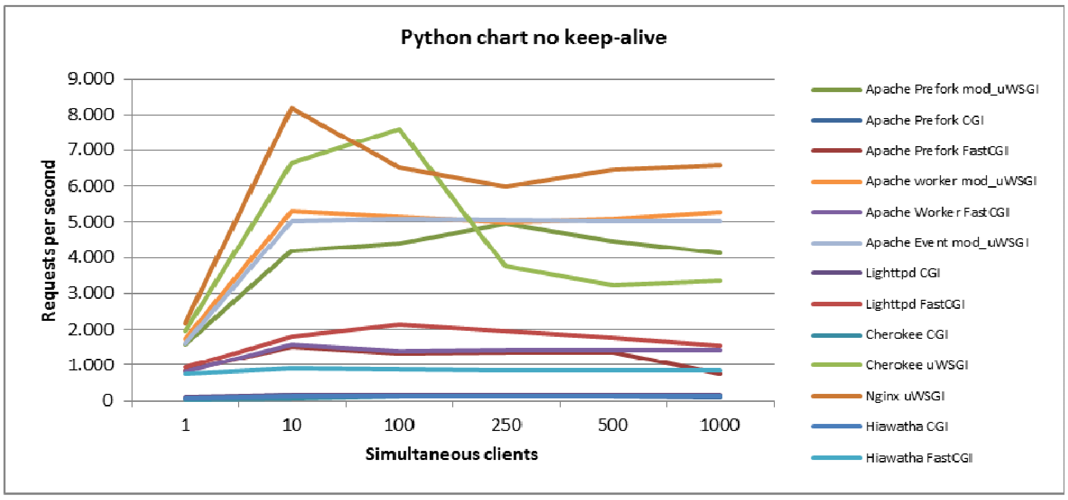

Fig. 4.16: Python no keep-alive chart

There are three different groups according to the obtained results shown in **Fig. 4.16.** The first one is composed by those using uWSGI interpreters, which are Apache, Cherokee, and Nginx. Performance of Apache web server is very stable, getting around 5,000 requests per second. Even Event configuration is more stable than Prefork. Nginx gets the best performance, reaching up to 8,000 requests per second, but as increasing the simultaneous clients in the system, the overall performance decreases, although it is maintained more or less stable.

Cherokee results shine another time. When increasing more than 100 simultaneous clients, the web server starts to decrease its performance. This drop in performance is not as high as in PHP test case, but makes us wonder why it happens and if it has some sort of solution.

The second group is formed by those web servers using FastCGI protocol to communicate between web server and Python interpreter. FastCGI is not as quick as uWSGI when serving the web page; nevertheless it gets 1,500 requests per second in average. Lighttpd does not have fully uWSGI module (being experimental for now), so it is only tested with FastCGI. Meanwhile, Hiawatha does only offer the possibility to use FastCGI. Al last, third group are those web servers that use CGI protocol. As happened with PHP test case, CGI performance is around 100 requests per second, which in comparison with other solutions such as FastCGI or uWSGI is very slow.

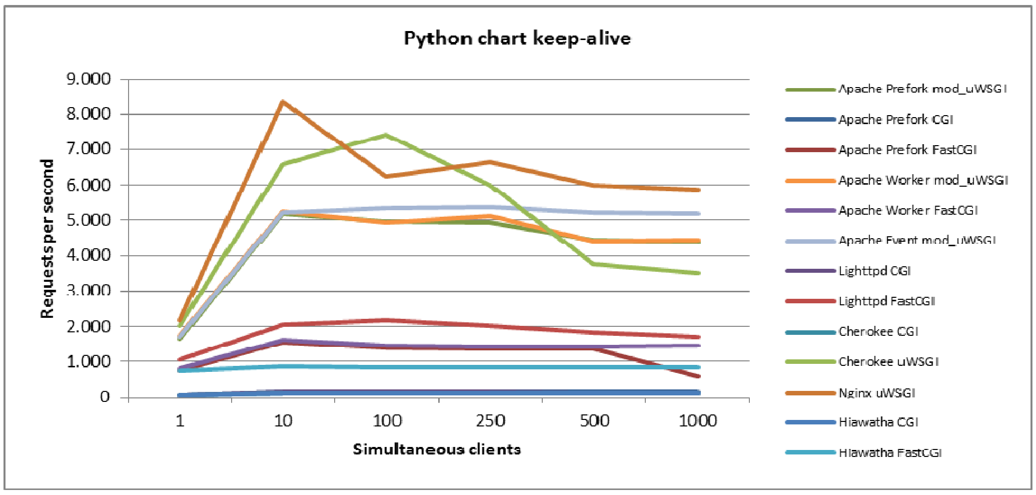

Fig. 4.17: Python keep-alive chart

Enabling keep-alive feature in python tests does not report any significant performance increase (see **Fig. 4.17**). There are also the three same groups than before, uWSGI, FastCGI and CGI. Cherokee performance does not have such a high decrease as before, but it still has a strange behavior.

Apache web server is often used to deliver dynamic web content. Furthermore it is being out for about 15 years, creating the possibility to use almost every dynamic language in different ways. During the Python test case, several interpreters are being used. **Fig. 4.18** shows the obtained results for Apache's Python test case. Enabling keep-alive feature does not represent a performance increase; instead, it shows some decrease in Worker uWSGI case. It is important to highlight that Apache Event configuration gets the best performance, although being in experimental mode yet.

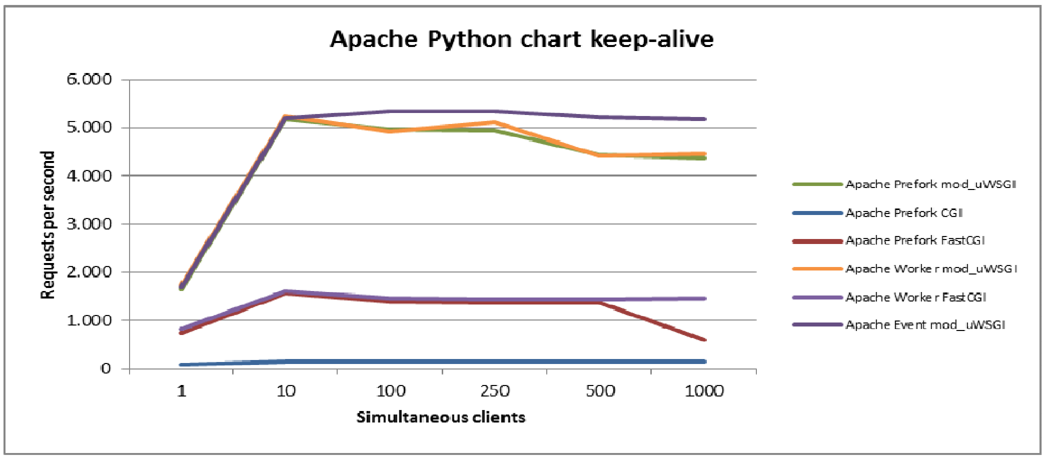

Figure 4.18: Apache Python keep-alive chart

Nginx web server also uses uWSGI module to run Python language. Although in previous charts it was shown that performance of Nginx's uWSGI was around 7,000 requests per second, it is interesting to see its scalability when dealing with multi process configuration. As in PHP test case, it is expected to see some variability in performance results. In Nginx tests it was only tested uWSGI interpreter, and the obtained results are very similar among the four configurations, see **Fig. 4.19**. Previous Nginx charts showed that two workers configurations was the best set up, although this does not apply here. Instead, the other configurations get better performance, even one worker configuration, which in previous tests was the slowest.

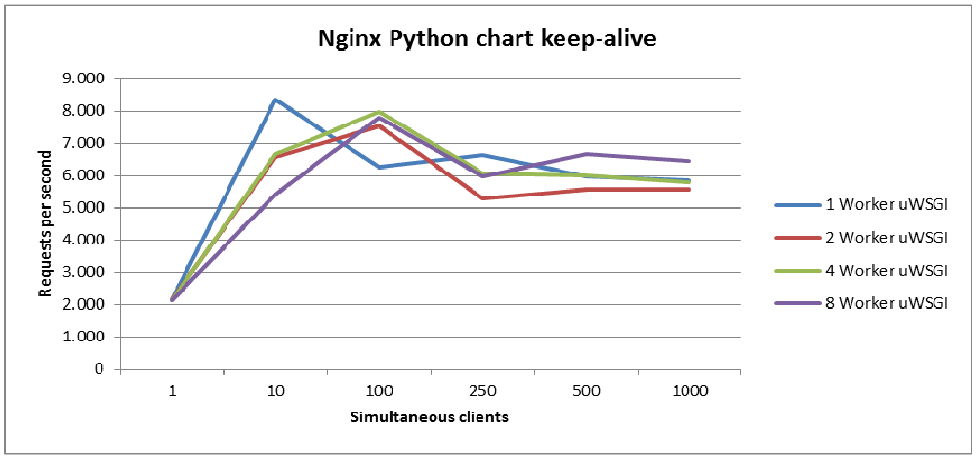

Fig. 4.19: Nginx Python keep-alive chart

Enabling keep-alive does not represent any performance improvement. Two worker configuration is the slowest of the four configurations but, as before, it is by a very small margin. Performance is around 6,000 requests per second when reaching 250 simultaneous clients in the system.

Lighttpd web server does not have uWSGI module for now, as it is being experimented without any special timeline to its official release. For this reason, it is tested in FastCGI and CGI protocol. As it is expected, FastCGI gets better performance results, see **Fig. 4.20**. As happened in previous tests, enabling keep-alive feature does not introduce any special performance increase. Multi process configurations get slightly better rates than single process configuration, although it does not decrease as much as before. Performance is around 1,800 requests per second in average.

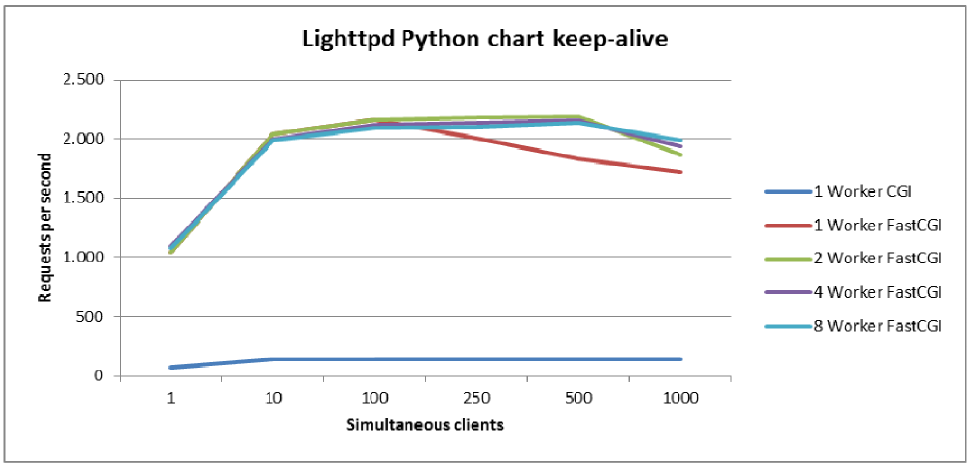

Fig. 4.20: Lighttpd Python keep-alive chart

4.2.3. Dynamic tests conclusions

At the beginning of the test case it was said that dynamic languages are more computational expensive to servers rather than static. So, performance could not be expected to be the same than for statics tests. PHP and Python languages were chosen to perform the dynamic languages test case because their importance in web development. Different web frameworks make use of those languages, such as Zend or Symfony use of PHP, meanwhile Django or Pylons use Python.

4.2.3.1. PHP tests conclusions

The first results obtained were PHP ones, setting some entrance point to compare the results among tested web servers. As happened with static tests, each web server behaves in its own way, meaning that although some of them share some kernel architecture, its performance rates are different. Also, not all of them offer the same configuration parameters, so it is important to understand those features and enable or edit them as convenience.

Apache httpd is a very popular web server, which could be primary used to deliver dynamic content in the web. There are different configuration parameters of it, as it was seen in static tests. Regarding the selected multi process configuration, some features could not be used. Using Apache Prefork configuration it was possible to enable mod_php, which is a kernel module add-on to Apache web server. Using this PHP kernel module performance increases a lot, reaching up to 10,000 requests per second with keep-alive enabled. Although this module is very useful when dealing with PHP language, it is not possible to use it when using Worker or Event configurations. Those configurations are multithread environment, and mod_php is not thread safe. To avoid further problems, Apache refuses to start when detecting mod_php enabled in the configuration file. The main drawback of mod_php is the memory footprint, because each Apache process has its own PHP interpreter embedded inside.

Not using mod_php means using FastCGI or CGI protocol. In previous chapters of this report, the differences of both protocols were stated. It was expected to see the main difference between both of them, which was performance. FastCGI, as its name says, is expected to be faster than CGI. In Apache httpd case, the three different configurations perform similar in FastCGI, reaching up to 3,000 requests per second. This rate is slower compared to mod_php performance. CGI is, in any case, slower, getting around 100 requests per second, which is a very slow rate.

Event-driven architecture web servers offer a very good performance, and also high scalability. It was proven to be very scalable in static tests, where the use of multi process environment helped the overall performance. Although, it could not be expected the same increase of performance due to the fact that it uses an external program, like FastCGI or CGI. Lighttpd FastCGI configuration is easy, as the same web server spawns the FastCGI processes. Lighttpd

FastCGI performance is higher than Apache's FastCGI, even when using single process configuration, which is near 4,000 requests per second. Configuring Lighttpd with more than one process shown some performance improvement, because it went from 1,000 to 1,500 requests per second more. Nevertheless, its performance is very similar for any configuration. Web server performance is hit by introducing more simultaneous clients in the system, with at least of 1,000 requests per second.

Nginx's configuration is very similar to Lighttpd's, although it needs to manually spawn the FastCGI processes. Being an event-driven architecture will give the ability to scale it by configuring it properly, the process is the same as for Lighttpd. Performance with one Nginx process is about 4,000 requests per second, which is similar to Lighttpd's. Using a multi process approach enables higher performance, getting up to 1,200 requests per second more. A good point of using the single worker configuration is its stability when reaching a high number of simultaneous clients. Other configurations may offer higher performance but they get a big hit when having 500 or 1,000 simultaneous clients.

Cherokee web server has a mix of sensations here. On one hand, it got a very decent result, with a rate around 4,500 requests per second, which is near Lighttpd and Nginx results. But on the other hand, it got an unexpected performance hit when introducing more than 100 simultaneous clients in the system. Performance in this case goes down to less than 100 requests per second. Tracking the load generated by Cherokee, it was possible to see that it was getting the 200% of CPU load, not spawning new FastCGI processes. And, after dealing with different configuration parameters, it was possible to see that configuring number of threads or the socket used, solves the issue (see **Appendix A.2**).

4.2.3.2. PYTHON tests conclusions

Within Python tests it was possible to see three main groups well differentiated by the technology that they are using. The most performing web servers are those using uWSGI interpreter, followed by FastCGI protocol, and finally CGI. Using uWSGI it was possible to top at 8,000 requests per second, although getting stable performance around 5,000 requests per second. FastCGI protocol was slower compared to uWSGI only having 2,000 requests per second. CGI protocol turned out to be the slowest performing protocol, something expected.

Apache httpd web server had plenty configuration options, as well as different modules to interpret python web pages. For Python tests it was tested with CGI and mod_uWSGI. Both cases could be used in the three different configurations of Apache. Apache mod_uWSGI got a performance around 5,000 requests per second, which was a very good value for the test. Also, it is important to highlight the performance generated by Worker or Event multi process configurations, which outperformed Prefork by a small margin. From the results point of view, Prefork module, which is the default configuration for Apache, it is not the best configuration for this type of tests.

For event-driven architecture web servers it is needed to split them, as Nginx supports uWSGI, while Lighttpd does not. Lighttpd FastCGI configuration was easy, and the obtained results were higher than Apache's, with 2,000 requests per second. By using a multi process environment configuration, it was not gained so much performance, but it maintained stable when increasing the number of clients above 250 simultaneous users. Lighttpd recommendation is to use a four worker configuration, and in this case it is good enough. Nginx's performance was very impressive, reaching top performance between 7,000 and 8,000 requests per second, which is one of the highest of the test. Varying the number of workers improves the stability when increasing the number of simultaneous clients, although it does not give any important performance increase. The recommended number of workers by Nginx's official documentation was two, but in this case, with even one worker it was found to be good enough.

Cherokee is again a mix of results. Given the default configuration letting Cherokee to choose the number of threads, by default is set to 10, gives very good performance until it reaches 100 simultaneous clients. Using TCP sockets does not solve this issue, as it has an overall slower performance. Given that this issue is very similar to what was found in PHP test cases, it was tested with different thread numbers (see **Appendix A.2**). There is a tradeoff between the number of threads and simultaneous users in the system. As threads increase better performance with less than 100 simultaneous clients, but as users increase better stay with least possible threads.

To conclude this section, it is possible to say that for Python web pages the best option is to choose an uWSGI interpreter, and pair it with Nginx, Cherokee or Apache. With these configurations it is ensured good performance.

4.3. HTTPS tests

Secure the web browsing is nowadays a must have feature. As more personal and confidential information is around the web, it is necessary to secure communications through it. Services like social networks, electronic payments or electronic mail make use of HTTPS for this reason. HTTPS uses SSL or TLS protocols to secure the communications by means of chains of certificates (PKI).

All web servers are configured to use ciphers with high security degree, but for comparison reason they were all set up to use *TLSv1/SSLv3 AES256-SHA*. When setting high ciphers, some web servers such as Apache and Nginx run the SSL protocol with *TLSv1/SSLv3 DHE-RSA-AES256-SHA*. Although, Lighttpd and Hiawatha can also run this security feature too, they refuse to do the SSL handshake when setting the latest cipher.

4.3.1. HTTPST-1 HTML test

This first test case consists in testing the performance of each web server in a secure environment by requesting an HTML page. The HTML web page is the same as for the first test, but this time accessed through HTTPS. The expectance of this test is to get far lower results than without SSL enabled. Also, the load generated into the CPU will be higher due to the fact that the web server needs to decrypt all the new incoming connections.

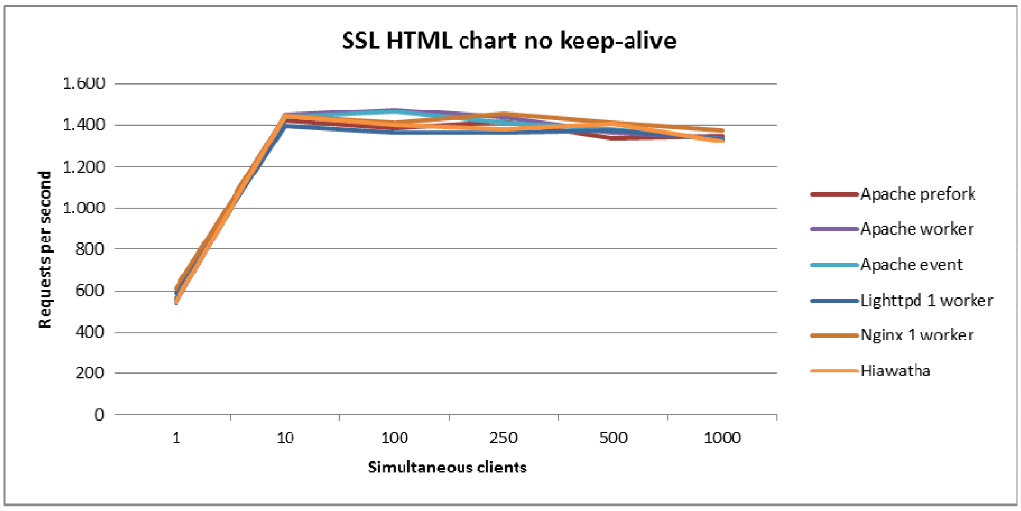

Fig. 4.21: SSL HTML no keep-alive chart

With keep-alive feature disabled during the test, it is possible to see that the performance of any of the tested web servers is equal to each other's (see **Fig. 4.21**). Performance tops around 1,400 requests per second. Although it is not shown in the figure above, increasing the number of workers in event-driven web servers, like Lighttpd or Nginx, does not increase the performance of the test (see **Appendix 2.3**). The obtained results are quite slow compared to the performance obtained when disabling SSL support. When enabling keep-alive feature of HTTP protocol, it is expected an increase of the performance of each one of the web servers; although, it will not be possible to reach, in any case, the performance obtained without SSL.

Fig. 4.22 shows the results of enabling keep-alive in all of the web servers. It is possible to see the increase of performance, going from 1,400 requests per second without keep-alive, up to 11,000 requests per second enabling keep-alive. In this case, the best performing web server is Apache using the Prefork configuration, which gets up to 11,000 requests per second, followed nearby by Worker's configuration. Apache's Event configuration performs lower than its two siblings, around 9,500 requests per second.

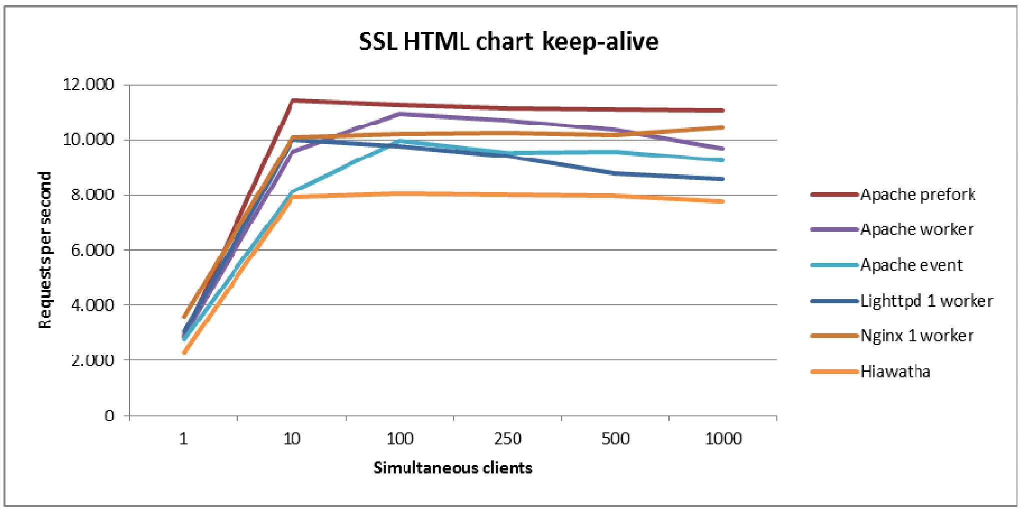

Fig. 4.22: SSL HTML keep-alive chart

Nginx web server has a very good performance, around 10,000 requests per second, very stable across the entire test. Meanwhile, the other event-driven web server, Lighttpd, suffers a decrease of performance when increasing the number of simultaneous clients in the system, with an average value of 9,000 requests per second. Hiawatha gets the lower performance, with around 8,000 requests per second.

On one hand, while not using keep-alive, the performance of event-driven web servers does not rise when increasing the number of active workers. But, on the other hand, using keep-alive results in a performance boosts for some configurations (see **Fig. 4.23**).

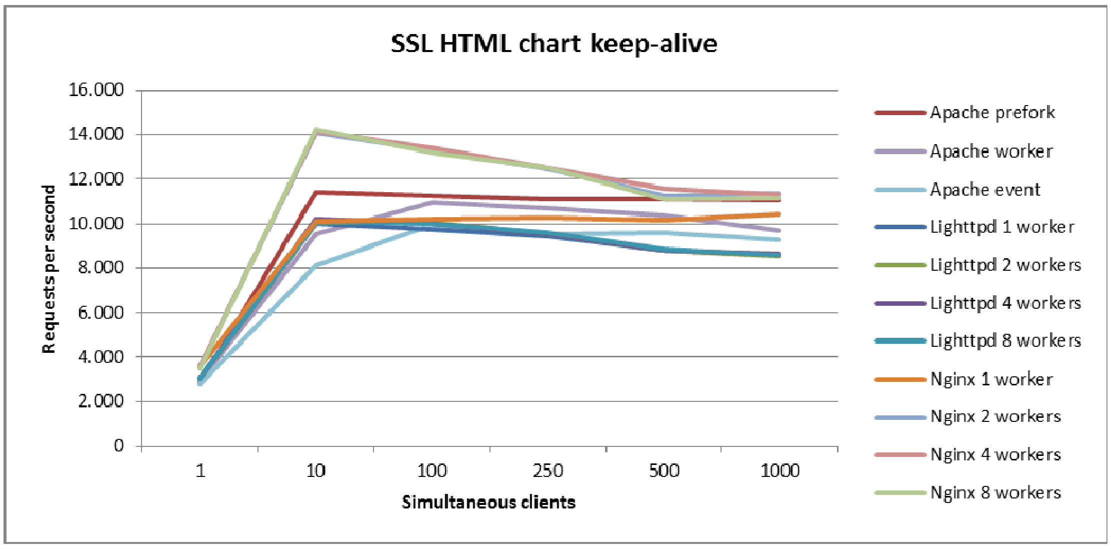

Fig. 4.23: Best performing SSL HTML keep-alive chart

In **Fig. 4.23** is compared the performance of using Apache web server (with different configurations) and event-driven configurations. In the previous chart, Apache Prefork configuration was the top performing web server, with 11,000 requests per second. Enabling different configurations for Nginx, changes a little bit the results. As it is possible to see, using two, four or eight simultaneous workers in Nginx, results in a performance increase, up to 14,000 requests per second, although it drops to 11,000 requests per second when reaching 1,000 simultaneous clients. Lighttpd configuration with multiple workers does not increase its performance; instead, it gets the same performance, no matter how many active workers are configured. So, Nginx's way of dealing with SSL sessions is better than Lighttpd's.

4.3.2. HTTPST-2 PHP test

As in the previous case, HTTPS HTML, it is expected some performance penalty when enabling HTTPS on PHP.

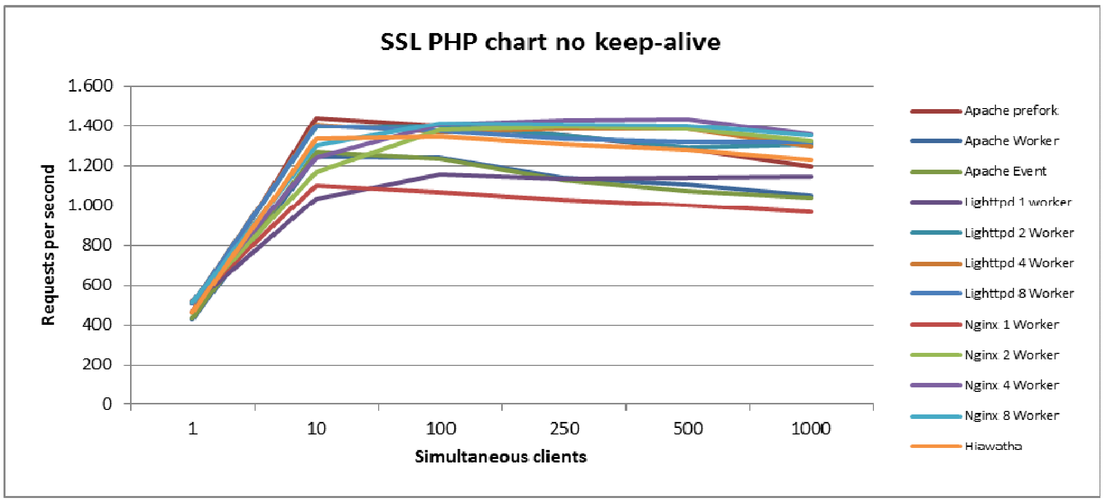

Fig. 4.24: SSL PHP no keep-alive chart

Regarding the use of SSL on PHP web pages, it is possible to see that the maximum performance is about 1,400 requests per second with keep-alive disabled (see **Fig. 4.24**). This is a very similar result that the one obtained from SSL HTML no keep-alive test, making us wonder if that is a performance limit of not using keep-alive when enabling SSL. The web servers that had a lower performance were Nginx and Lighttpd default configuration, as well as Apache's Worker and Event configuration. Meanwhile, configuring Lighttpd and Nginx to use more than one active worker boosts performance of the web server, getting around 1,300 and 1,400 requests per second, an increase of 30%.

When enabling keep-alive, the chart (**Fig. 4.25**) got divided in three different groups. The first group is formed by Lighttpd's multi process configuration, getting up to 4,500 requests per second, which is a raise of 221%. In second group is Lighttpd's default configuration, which is around 3,000 requests per second. And, in third group are the rest of web servers. In this case, Nginx multi process configuration does not help to increase the web server's performance; instead, it does not change, with 1,400 requests per second. Apache also got the same performance as Nginx's, with around 1,200 requests per second.

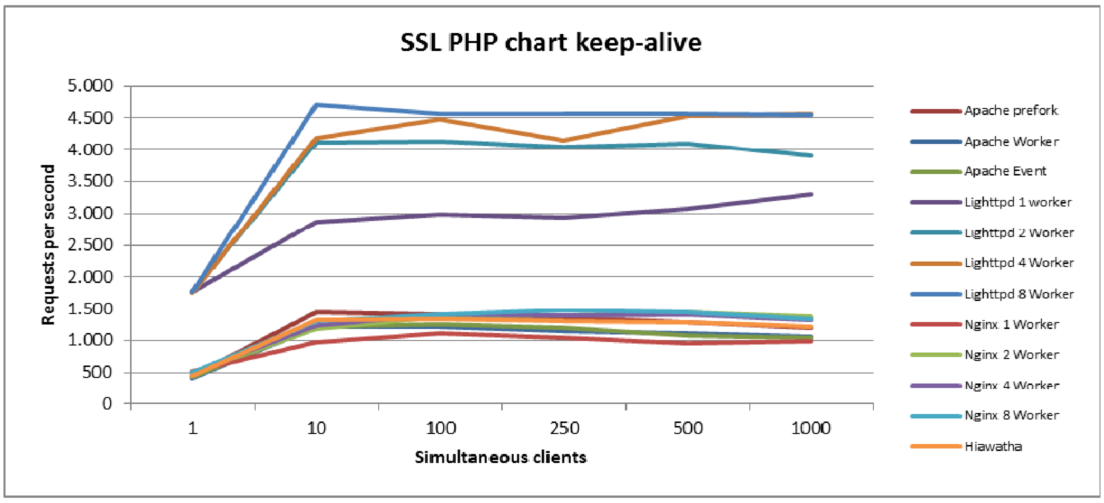

Fig. 4.25: SSL PHP keep-alive chart

4.3.3. HTTPST-3 PYTHON test

Performance of HTTPS Python is similar to the previous test, around 1,400 requests per second in the best case. In the first scenario, without keep-alive, it is possible to see four different groups (see **Fig. 4.26**). The first group is formed by Apache web server in its different configurations, which had a good performance, around 1,400 requests per second. Prefork configuration starts to decrease its performance when reaching 500 simultaneous clients, although the other two configurations remain stable.

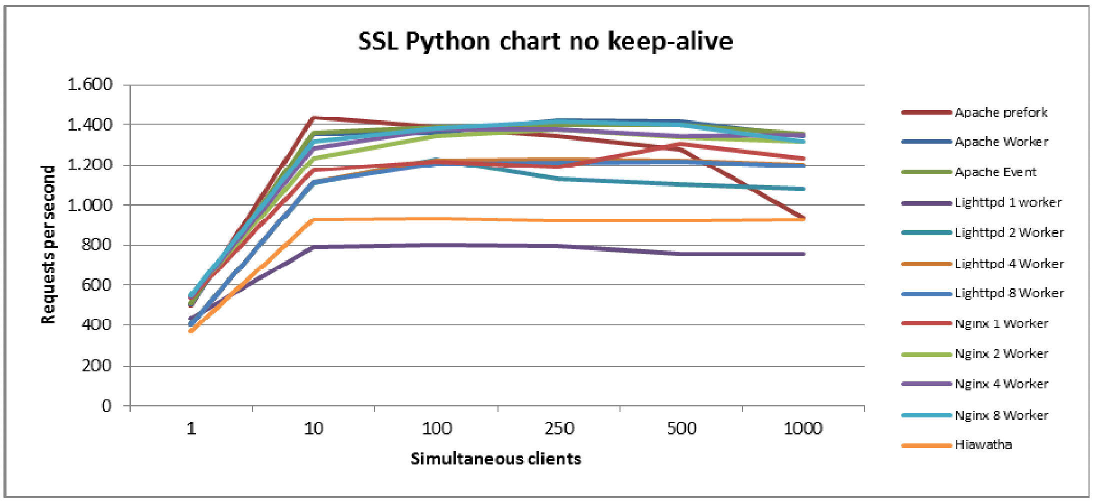

Fig. 4.26: SSL Python no keep-alive chart

Nginx's configurations with more than one active workers also matches Apache's performance, around 1,400 requests per second. The second group is composed of Nginx default configuration (one active worker) and Lighttpd multiple worker configuration. In the case of one Nginx worker, its performance remains stable up to 1,200 requests per second, although it increases a little when raising the number of simultaneous clients. The third group is form by

Hiawatha, which runs SSL Python with 900 requests per second. Finally, the latest group is formed by Lighttpd, running its default configuration with only one worker.

Enabling keep-alive does not produce any considerable increase, only moving from 1,400 up to 1,900 requests per second, both in the best case (see **Fig. 4.27**). Here there are two different groups; the first one is formed by Lighttpd multi worker configuration, which outperforms its default configuration. In the second group are the rest of the web servers that are Apache, Lighttpd default configuration, Nginx, and Hiawatha. All of them have a similar performance than in the previous test, without keep-alive enabled, which were around 1,400 requests per second.

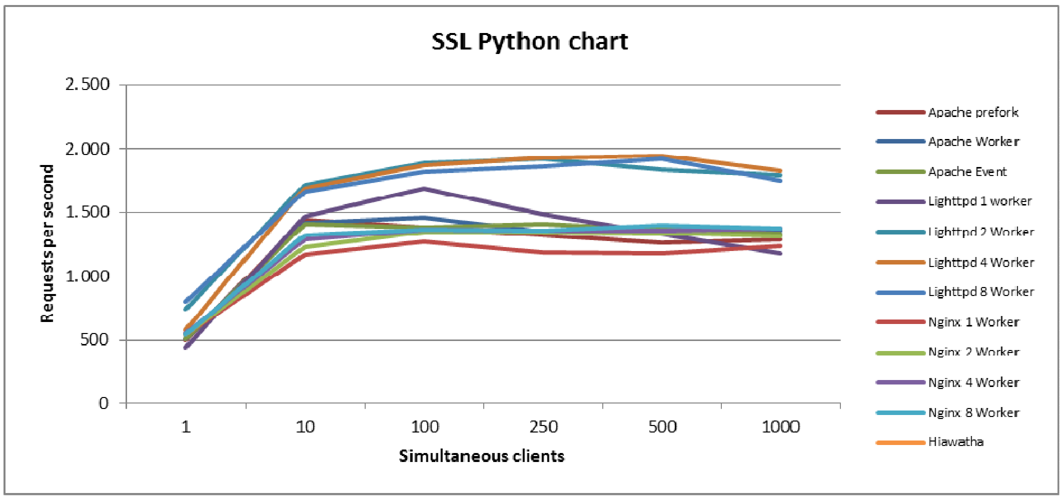

Fig. 4.27: SSL Python keep-alive chart

4.3.4. HTTPS tests conclusions

After conducting HTTPS tests, it is possible to say that the performance penalty of enabling such feature is high. For this test it was chosen AES256-SHA1, which is marked as high by OpenSSL, but there are even more secure communications. Eventually, this will affect the overall performance of the web server, having it even more load and with less simultaneous clients.

4.3.4.1. HTTPS HTML test conclusions

During the HTTPS HTML tests it was obvious that exists a clear performance difference between enabling SSL or keep-alive. When using SSL without keep-alive performance of any web server was limited to 1,400 requests per second, which was a very low rate compared with those gotten from the normal HTML test. No matter the web server architecture which is being deployed or even if they had more than one active process at the same time, performance was limited. Once enabling keep-alive, those rates increased up to 14,000 in the best case, but still far from no SSL enabled results.

Performance penalty was dependent of each web server. Comparing Apache's performance in the different test cases, it is found a difference between 480% and 680% regarding its configuration. Lighttpd performance without SSL was slightly higher than Apache's, so the performance difference by enabling SSL increases too, between 530% and 680%, regarding the different worker configurations. Nginx's performance was similar to Lighttpd, having a performance penalty between 460% and 650%, regarding its configuration.

Once enabled keep-alive, there is an increase of the performance. Each web server increases its performance, but nevertheless reaching the rates obtained without SSL. In Apache's case, it rises from 1,400 requests per second up to 8,000 – 11,000 requests per second, but has a performance penalty around 67% by using SSL. Lighttpd increases its performance up to 10,000 requests per second, but taking into account its static; no SSL performance is a penalty between 30% and 60% regarding its configuration. Nginx's keep-alive test shows an increase of performance, going from 1,400 requests per second up to 10,000 – 14,000 requests per second, but the overall performance penalty of enabling SSL is between 42% to 50% regarding its different configurations. See **Appendix 3.3.1** to view more performance results.

Using SSL reduces the performance of the web server between 30% and 60%, which has a high impact on the web server itself. This is translated to a higher CPU load, as well as smaller amount of simultaneous users in the system and lower performance. If SSL performance is an important objective, it may be needed to use a multiple server configuration with a load balancer.

4.3.4.2. HTTPS PHP tests conclusions

As for the previous test, enabling HTTPS to PHP web pages provokes a reduction of the web server's performance. Regarding the technology associated to each execution environment as well as the architecture of the web server, could be possible to increase the number of requests per second when enabling keep-alive feature.

HTTPS PHP test case showed to have a low performance rate without keep-alive having around 1,400 requests per second, using a "Hello, World!" script. Keep-alive feature introduces some performance increase for Lighttpd, reaching up to 4,000 requests per second. Although, other web server such as Apache do not benefit of kernel loaded extensions, making it to perform like Nginx or Hiawatha. So, this leaves space to think about what would happen when requesting even larger web pages, with more dynamic content. For this test, the clear winner is Lighttpd, which gets higher performance than the others. See **Appendix 3.3.2** to view more performance results.

4.3.4.3. HTTPS PYTHON tests conclusions

The latest HTTPS test case consisted in requesting a HTTPS Python web page. Performance associated to this test was similar to the previous ones, the HTML and PHP HTTPS tests. Without keep-alive enabled, the performance of almost all web servers was between 1,200 and 1,400 requests per second. Lighttpd and Hiawatha have lower than expected performance, but it is not a major special problem as it is solved in the next scenario.

When enabling keep-alive, it appear two groups. Any of them had a high performance increase, having in the best case up to 1,900 requests per second. Any of the tested web servers had a high performance advantage over the others; only Lighttpd's seems to achieve higher rates. So, in this case, Lighttpd is the winner of this test case, by a small margin. As in HTTPS PHP test case, the performance associated to this scenario is relatively small, needing more powerful machines or even clusters in order to serve thousands of requests. See **Appendix 3.3.3** to view more performance results.

4.4. Load test

The final test case consists in generating a continuous load from different clients to request some web page. As explained in previous chapters of this report, it will be done by using a distributed benchmarking tool, which will distribute the load between different clients. Also, it will be useful in order to avoid some operating system limitations, which afterwards could affect the test results. Until now, all tests were requests per second performance centric, now it is time to get deeper and show how many simultaneous clients the web server can handle. It is going to use MySQL and PHP, so both will have a high performance impact in the web server, no matter that it is requesting the default main page.

Load test will consist in creating three different phases during 1 hour time frame. The first phase consists in generating 100 clients per second, during 30 minutes. The second phase will create 450 clients per second during 10 minutes, and the last phase 200 clients per second during 20 minutes. Each client will do two requests, one to the main page and the second to the example post. There will be a 5 second of sleep time between requests. The second phase is the most demanding, as it loads the processor up to the limit, and also increasing the RAM usage. If the web server is not efficient, some timeout errors or server crashing may occur.

For this test it was selected only five of the tested web servers, as they offered the best performance during previous tests. Those web servers are Apache Prefork and Event configurations, Lighttpd, Cherokee and Nginx. Others like Mongrel2, Yaws, Hiawatha or Tomcat were discarded because of their lower performance. Lighttpd was configured to use four active workers, meanwhile Nginx to use two workers.

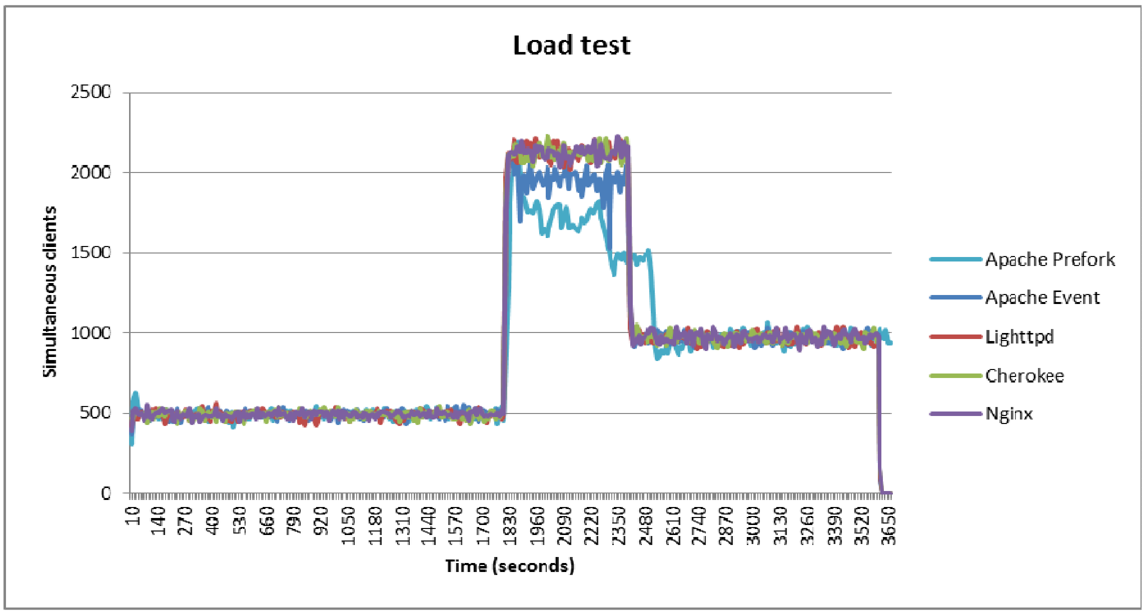

Fig. 4.28: Capacity test performance chart

Fig 4.28 shows the obtained results during the load test. Apache Prefork configuration does not perform as expected, and has some performance issues during the second phase. In this case, it had both full processor load and RAM memory usage, lowering the performance and generating some timeout errors. Meanwhile, Apache Event configuration got better performance than Prefork, although slightly lower than for Lighttpd, Nginx and Cherokee. Event configuration did not run out of RAM memory, although it reached the full CPU load.

Lighttpd, Nginx and Cherokee did get the same performance, not having any issue with any of the three launched phases. CPU load during the second phase was high, up to 90%, although lower than for Apache's configuration. RAM memory consumption depended of each web server, Lighttpd RAM footprint was the lower, only consuming 2GB, which is the 50% of the available RAM; whereas Nginx memory consumption was higher, up to 2.8GB. Cherokee consumed even more RAM memory, 3.3GB. Although, neither of those web servers run out of memory or CPU, having still some space for scalability (with higher load phases).

In order to go deeper into some information such as CPU load or RAM memory footprint see **Appendix 3.7.**

CHAPTER 5. LESSONS LEARNED

Configuring a web server is not an easy task. As diving deeper into settings and performance, is possible to see that there are different parameters to take into account in order to improve the overall performance, as well as reduce some possible issues. During this project, some parameters of either the web server or the operating system had been tuned in order to offer better performance. On the operating system side, it was tuned the ***ulimit*** directive, which establishes the number of file descriptors that any program can open during its operation. By default, the maximum value is 1024, which in some cases with high volume of transactions, it may be not enough making the web server to crash or limit the performance. So, to have more file descriptors, thru limit directive was tuned up to 65535, enough for the tests that were made.

Another complicated task is to give correct file and folder permissions. It is not as easy as coping the files and start using the web server. Consequently, it is important to realize which are the correct file permissions and apply them. The clearest example that occurred during the project was when using Wordpress blog. In that case, it was needed to set the correct ***chmod*** directive in order to allow the web servers user to change, add or remove files. The very basis values for this directive were 644, 755, 775 and 777 (although it may not be recommended).

Also, each web server has its own parameters to configure. Talking about Apache, it was interesting to see the difference in the performance when enabling the three possible configurations. The default one, Prefork, does give the best performance, giving the option to load PHP language into the kernel by means of a module. But, it has its own drawbacks, as it creates a new process for each new incoming request, limiting the overall performance of the web server. Moreover, using the PHP modules increases its performance, but loads a PHP interpreter into each process, increasing the memory footprint. The other configurations help to have a wider vision of the different architectures. For instance, Event configuration, introduces in to the event-driven architecture, in which a lot of new applications are deploying its efforts.

Event-driven architectures are driving the next evolution of software development, due to the fact of the different performance enhancement that produces such architecture. For this reason, Lighttpd and Nginx web servers are best suited than others for some specific tasks. Event-driven architectures make use of non-blocking I/O directives, which makes possible to perform well with only one process and one thread. But, in order to increase its performance, it is advisable to fork the main process into more. Both cases benefit from such configuration, leading to a noticeable performance boosts.

Multithreading web servers such as Cherokee or Tomcat also have high performance, due to the full usage of the processor. In this case, the capability of creating as many threads as necessary loads completely the CPU. But, also both have their own drawbacks, being the main one that not all the applications are thread safe, leading to a possible issue when there is some memory share. Speaking about Cherokee, it has a very good performance as well as a very intuitive web configuration, which makes easier to configure. On the other hand, Tomcat offers very good performance, but it is mainly used as an application server, not integrating FastCGI. This might seem a drawback, but its strength is the compliance with Servlets and JSPs.

As explained in previous sections, FastCGI, that replaces CGI, has a very good performance and compatibility for almost all dynamic languages. PHP languages benefit from FastCGI execution by not creating a new process for each new incoming request. Also, it is important to set the proper FastCGI configuration parameters, such as number of processes and requests per process. Those parameters will affect the overall performance as well as the memory footprint. Setting a small value for both could impact negatively the performance of the web server, as not all the requests could be attended quickly. Meanwhile, setting it to a high value might load too much the web server, leaving no memory for other processes. The other dynamic language tested during this report was Python. As PHP, Python scripts needs an external interpreter to be executed. There are different interpreters, but it was chosen uWSGI. There were different configuration parameters of uWSGI software, and it was needed to change them according to their performance. Avoiding some logging, as well as creating sub processes, allows increasing the performance offered by such interpreter.

The last part of the report was focused on having real performance numbers. Those tests where done by using a free framework applied to make blogs in the Internet. Such tool gives the possibility to know the performance associated to a machine when using a real workload framework. Also, it was installed some caches in order to boosts the web server’s performance, like in the real world. For this reason, it was installed some PHP accelerators, in order to accelerate the PHP code that was present in the Wordpress blog by converting it to op-code.

In addition, for any of the performed tests, the requests per second values obtained are important. Tested web pages are too small to be considered real web pages, as Internet frameworks make use of different types of languages as well as interaction with other software, such as databases. The main reason behind the use of such small web pages is to know the best performance associated to each one of the web servers. Having both the hardware and the web pages equal among the web servers, gives the opportunity to explain performance differences from their architectural differences.

Watching closely the error logging file as well as the CPU and the RAM memory performance is mandatory. Knowing those three aspects of the server gives an idea of how it performs, and if there are any issues during its execution. Avoiding any issue and configuring each web server according its possibilities is difficult, but it is needed in order to get the best experience of it. Getting a web server to work is an easy task, but making it perform well and take the best in any case is a hard job.

CONCLUSIONS

Web servers are a very important Internet tool, as they are in charge of deliver content for any web page that is visited. Usage of Internet is being increased day-by-day in order to introduce even more services, which once upon a time were provided by physical persons. At the beginning of this thesis, some important features were introduced, in order to give a general idea of what was needed to be considered and what was the motivation of the work.

When benchmarking any product, it is important to define as well as possible the whole scenario and the test cases to benchmark (so it can be shown the strengths and weaknesses of any web server). Scenarios consisted in requesting different web pages technologies such as static and dynamic pages, and also secure connections. The first part of the results were static tests, where it was tested an HTML web page and two different sized images. Those three tests got the highest performance rates, as they used fewer server resources not needing any special processing.

Dynamic content tests consisted in serving dynamic languages such as PHP, Python and Java. Those languages are widely deployed on the Internet, and used by lot of web frameworks. Their performance is lower, due to the fact that require external interpreters in order to process the different languages. Moreover, using external interpreters, gave the option to test different dynamic web technologies such as CGI, FastCGI, Servlet, JPS, and uWSGI. Those protocols were benchmarked and compared against each other, to see their relative performance. Secure tests consisted in enabling the use of SSL in the web servers, and the use HTTPS protocol instead of HTTP. Enabling such feature had a high performance penalty when compared with not using SSL. The last test consisted in creating a load scenario where simultaneous clients in the system were benchmarked.

Any benchmark depends on what it is going to be tested. In order to benchmark web servers it is important to have the whole web site or the used framework; although, it is not always possible. In the meantime, using small web pages gives a clear idea of the maximum performance that any web server can deliver, letting the user to decide what is the best for any situation. After conducting such tests, it is possible to say that for static content Lighttpd and Cherokee were the best web servers. For dynamic web pages, servers like Apache httpd, Nginx and Cherokee had the best performance, regarding the used technology. Speaking about secure communications, Lighttpd, Apache and Nginx offer very good performance against the others.

There is not the perfect and best suitable web server for all cases; although, some of them can be the best in some specific scenarios. So, high load cases will require the use of more than one web server, to distribute functionalities according to their strengths.

REFERENCES

[1] Internet World Stats. http://www.internetworldstats.com/stats.htm

[2] Netcraft Internet statistics. http://news.netcraft.com/

[3] Telefónica I+D. http://www.tid.es

[4] Tim Berners Lee. http://www.w3.org/People/Berners-Lee/

[5] Where the web was born, CERN (European Organization for Nuclear Research). http://info.cern.ch/Proposal.html

[6] HTTP protocol. http://www.w3.org/Protocols/

[7] HTTP 0.9 protocol. http://www.ietf.org/rfc/rfc1945.txt

[8] HTTP 1.0 protocol. http://www.ietf.org/rfc/rfc1945.txt

[9] HTTP 1.1 protocol. http://www.ietf.org/rfc/rfc2616.txt

[10] Common Gateway Interface (CGI). http://www.ietf.org/rfc/rfc3875

[11] FastCGI, Open Market. http://www.fastcgi.com

[12] Java Servlet technology, Oracle. http://www.oracle.com/technetwork/java/javaee/servlet/index.html

[13] Java Server Pages, Oracle. http://www.oracle.com/technetwork/java/javaee/jsp/index.html

[14] WSGI. http://wsgi.org/wsgi/WsgiStart

[15] Python Web Server Gateway Interface v1.0, PEP 333. http://www.python.org/dev/peps/pep-0333/

[16] uWSGI. http://projects.unbit.it/uwsgi/

[17] Apache httpd web server, Apache Foundation. http://httpd.apache.org/

[18] Lighttpd web server. http://www.lighttpd.net/

[19] Cherokee web server, Octality. http://www.cherokee-project.com/

[20] Nginx web server. http://nginx.org/

[21] Web servers to handle ten thousand clients simultaneously, The 10K problem. http://www.kegel.com/c10k.html

[22] Mongrel2 web server. http://mongrel2.org/home

[23] Hiawatha web server. http://www.hiawatha-webserver.org/

[24] Tomcat web server, Apache Foundation. http://tomcat.apache.org/

[25] Yaws web server. http://yaws.hyber.org/

[26] Iperf network benchmark tool. http://sourceforge.net/projects/iperf/

[27] Apache benchmark tool, Apache Foundation. http://httpd.apache.org/docs/2.0/programs/ab.html

[28] Tsung benchmarking tool. http://tsung.erlang-projects.org/

[29] Wordpress blog site. http://wordpress.com/

[30] Ganglia monitoring system. http://ganglia.sourceforge.net/

[31] Welsh, Matt; The Staged Event-Driven Architecture for Highly-Concurrent Server Applications; Computer Science Division University of California, Berkeley.

[32] Li, Peng; Zdancewic, Steve; Combining Events And Threads For Scalable Network Services; University of Pennsylvania.

[33] Rajagopalan, Mohan; T. Lewis, Brian; A. Anderson, Todd; Thread Scheduling for Multi-Core Platforms; Programming Systems Lab, Intel Corporation.

[34] Gammo, Louay; Brecht, Tim; Shukla, Amol; Pariag, David; Comparing and Evaluating epoll, select, and poll Event Mechanisms; University of Waterloo.

[35] Nørmark, Kurt; Using Lisp as a Markup Language, The LAML Approach; Department of Computer Science of Aalborg University.

[36] History of the Internet. http://www.isoc.org/internet/history/cerf.shtml

APPENDIX

A.1 Scenario configuration

In **Chapter 3** was explained the main hardware and software specifications that have the available machines. In the first appendix of the thesis, the **A.1**, it will be shown with more detail some of the hardware and software configurations, as well as the web server's configuration files.

A.1.1 Used hardware and software

A.1.1.1 PC1

This machine is the main computer, where all the scripts and processes are programmed, along with all the generated data during the tests. It is also used to connect remotely to all the servers/clients of the network though SSH.

Table A.1: PC1 Hardware

PC1 Hardware	
Processor type	Intel Pentium D 820 (Rev. B0)
Processor cores	2
Processor frequency	2.80 GHz
RAM memory	3 GB DDR2-5300 (@ 667 MHz)
HDD	80 GB @ 7200 rpm SATA
Network Interface Card	Broadcom NetXtreme 57xx Gigabit Controller @ 1000 Mbps
Operating system	Microsoft Windows 7 (x86)
Java	1.6.0_23
PHP	5.3.6
Python	2.6.4
Apache	2.2.17
MySQL	5.1.41

Table A.1 shows the hardware and installed software of the PC1. This is a Microsoft Windows machine. It will give us the opportunity to use almost all of the programs that are going to be tested, but also some performance issues compared with Linux machines thinking as a server machine.

A.1.1.2 TAS01

This machine will be used as a main server machine, where it will be running the different Web Servers that are going to be tested. It has almost the same hardware than the previous computer, but the software is slightly different.

Table A.2: TAS01 Hardware

TAS01 Hardware	
Processor type	Intel Pentium D 950 (Rev. C1)
Processor cores	2
Processor frequency	3.40 GHz
RAM memory	4 GB DDR2-5300 (@ 667 MHz)
HDD	250 GB @ 7200 rpm SATA
Network Interface Card	Broadcom Tigon3 BCM5751 Gigabit Controller rev 4001 @ 1000 Mbps
Operating system	Red Hat Enterprise Edition 5.3 (x86_64)
Java	1.6.0_14
PHP	5.3.6
Python	2.4.3 and 2.6.5
MySQL	Ver 14.12 Distrib 5.0.45

Table A.2 shows the hardware and software specs of the TAS01 server. As seen, it is better than the PC1 because has a more powerful CPU, but probably this is not a key aspect due to the fact that both machines share the same architecture of Intel Pentium D. The main difference between TAS01 and PC1 is that this computer runs a Linux distribution running a 64bits kernel, which could lead to a better performance in a server point of view if compared with the PC1.

A.1.1.3 TAS02

This machine will be used as a backup machine for different uses. It will work as a server machine; it has the same web servers than those found in TAS01. It could be configured to create a cluster of machines to increase the performance of the whole web server structure. Also, it will be used as a client, providing a machine to perform the benchmark tests and also to perform a distributed benchmark with other machines.

Table A.3: TAS02 Hardware

TAS02 Hardware	
Processor type	Intel Xeon 5110 (Rev. B1)
Processor cores	2
Processor frequency	1.60 GHz
RAM memory	4 GB DDR2-5300 (@ 667 MHz)
HDD	250 GB @ 7200 rpm SATA
Network Interface Card	Broadcom Tigon3 BCM5751 Gigabit Controller rev 4201 @ 1000 Mbps
Operating system	Red Hat Enterprise Edition 5.5 (x86_64)
Java	1.6.0_14
PHP	5.3.6
Python	2.4.3 and 2.6.5
MySQL	Ver 14.12 Distrib 5.0.45

Table A.3 shows the specs in hardware and software of the machine TAS02. The architecture of the CPU, an Intel Xeon 5110, is similar to the one found on PC1 and TAS01, a Pentium D. The main software specs are the same than for TAS01, only changing the operating system from the 5.3 version to the 5.5.

A.1.1.4 TAS03

This machine will be used as a backup machine for different uses. Will work as a server machine, where it will be running the same web servers than those found in TAS01. It could be configured to create a cluster of machines to increase the performance of the whole web server structure. Also, it will be used as a client, providing a machine to perform the benchmark tests and also to perform a distributed benchmark with other machine.

Table A.4: TAS02 Hardware

TAS02 Hardware	
Processor type	Core 2 Duo 6320
Processor cores	2
Processor frequency	1.86 GHz
RAM memory	3.5 GB DDR2-6400 (@ 800 MHz)
HDD	750 GB @ 7200 rpm SATA
Network Interface Card	Realtek RTL8168b/8111b @ 1000Mbps
Operating system	Red Hat Enterprise Edition 5.5 (x86_64)
Java	1.6.0_14
PHP	5.3.6
Python	2.4.3 and 2.6.5
MySQL	Ver 14.12 Distrib 5.0.45

TAS03 hardware is better than the previous ones, although it was no available at the beginning of the test cases. It is be very useful when dealing with distributed tests, in which is needed some extra computational power.

A.1.1.5 TAS04

This machine will be used as a client machine, where the different benchmarking tools will be running. Also, it offers the possibility to create a distributed benchmark using the process capacity that offers the TAS02 and TAS03 machines.

Table A.5: TAS04 Hardware

TAS04 Hardware	
Processor type	Intel Pentium 4 521 HT
Processor cores	1 (with Hyper Threading)
Processor frequency	2.80 GHz
RAM memory	3 GB DDR2-5300 (@ 667 MHz)
HDD	80 GB @ 7200 rpm SATA
Network Interface Card	Broadcom Tigon3 BCM5751 Gigabit

	Controller rev 4001 @ 1000 Mbps
Operating system	Red Hat Enterprise Edition 5.3 (x86_64)
Java	1.6.0_14
PHP	5.3.6
Python	2.4.3 and 2.6.5
MySQL	Ver 14.12 Distrib 5.0.45

Hardware specs for the TAS04 (see **Table A.5**) machine are lower than for the other. While the hardware of the machine changes, the software is the same as those found in the TAS01, sharing all the same application versions. Benchmarking tools do not load so much the CPU; so, using a lower specs machine is not critical from the client's point of view.

A.1.2 Network and operating system configuration

The network and the operating system plays an important role during any benchmark as it is the main layers where the service will be running. Having good connectivity is important to avoid some bottlenecks generated which will limit the performance. If the system is limited by network bandwidth there is little to be done, if not changing the network infrastructure. On the other hand, the operating system will limit the available resources in the machine, as it has all the directives to share with other running processes.

A.1.2.1 Network bandwidth

In order to test the network bandwidth it is used Iperf. Iperf tool is an open source network benchmarking tool, where it is possible to set a client and a server and start sending data between them. To test the available network bandwidth between the different servers, it will be installed in the four server machines and complete very simple tests between them.

- To start an Iperf server side, it is only needed to look for the program and run it, like this: **iperf –s**. This will open the default TCP port and wait for receive data.
- To start the Iperf client side, it is needed to look for the program and run it, like this: **iperf –c tas01 –P4**. This will start four clients and start sending data to the default TCP port on TAS01 server.

Table A.6: Results of network benchmarking:

Server / **Client**	**TAS01 (Mbps)**	**TAS02 (Mbps)**	**TAS03 (Mbps)**	**TAS04 (Mbps)**
TAS01 (Mbps)	-	945	945	948
TAS02 (Mbps)	947	-	947	945
TAS03 (Mbps)	890	891	-	891
TAS04 (Mbps)	949	948	946	-

After conducting these network tests (see **Table A.6**), it is possible to say that the network works in a gigabit mode, allowing us to maximize the network testing and performance. Having a gigabit network (very common in most of the cases) will become the bottleneck in cases of high-sized file transfers where is need a high bandwidth. In other cases, the CPU or the RAM of the machines will be the bottleneck of the system.

A.1.2.2 Operating system

Here are some of the parameters that are configured in the different machines. As said in previous chapters of this report, these machines are Linux. They have Red Hat Enterprise Linux version 5.3 or 5.5. These parameters make reference to the operating system buffers as well as TCP/IP protocol. Those are found in **/proc/*sys/net/core/*** and **/proc/*sys/net/ipv4/***.

Table A.7: TCP/IP parameters for Red Hat web servers

Parameters		**TAS01**	**TAS02**	**TAS03**	**TAS04**
rmem_default	Default OS receive buffer size	126976	129024	129024	126976
rmem_max	Max OS receive buffer size	131071	131071	131071	131071
wmem_default	Default OS send buffer size	126976	129024	129024	126976
wmem_max	Max OS send buffer size	131071	131071	131071	131071
ipv4.tcp_rmem	TCP Autotuning setting Receive buffer	4096 87380 4194304	4096 87380 4194304	4096 87380 4194304	4096 87380 4194304
ipv4.tcp_wmen	TCP Autotuning setting Send buffer	4096 16384 4194304	4096 16384 4194304	4096 16384 4194304	4096 16384 4194304
ipv4.tcp_mem	TCP Autotuning setting	196608 262144 393216	196608 262144 393216	196608 262144 393216	196608 262144 393216
tcp_timestamps	Timestamp add 12 bytes to the TCP headers	1	1	1	1
tcp_dsack		1	1	1	1
tcp_sack	TCP selective acknowledgments	1	1	1	1
tcp_window_scaling	Support for large TCP Windows	1	1	1	1
ip_forward		0	0	0	0
tcp_fin_timeout	TCP connection timeout	60	60	60	60
tcp_keepalive_time	TCP connection keep alive time	7200	7200	7200	7200

As seen in **Table A.7**, almost all parameters are the same for the four machines. Also they have very similar software configurations, so the machines will be used in this scenario are very similar, not having any special tweaking to perform better than others. But, it is important to know what they mean, for this reason is made a brief summary of those directives (source: http://www.linuxinsight.com/proc_sys_hierarchy.html).

- **Rmem_default**. The default setting of the socket receive buffer in bytes.
- **Rmem_max**. The maximum receive socket buffer size in bytes. The default value is 131072 bytes.
- **Wmem_default**. The default setting of the socket send buffer in bytes.
- **Wmem_max**. The maximum send socket buffer size in bytes. The default value is 131072 bytes.
- **Ipv4.tcp_rmem**. Vector of 3 integers: min, default, max.
  - Min - minimal size of receive buffer used by TCP sockets. It is guaranteed to each TCP socket, even under moderate memory pressure. The default value is 4096 bytes.
  - Default - default size of receive buffer used by TCP sockets. This value overrides rmem_default used by other protocols. The default value is 87380 bytes. This value results in window of 65535 with default setting of tcp_adv_win_scale and tcp_app_win is 0, and a bit less for default tcp_app_win.
  - Max - maximal size of receive buffer allowed for automatically selected receiver buffers for TCP socket. This value does not override rmem_max, "static" selection via SO_RCVBUF does not use this. The default value is 4194304 bytes.
- **Ipv4.tcp_wmem**. Vector of 3 integers: min, default, max.
  - Min - amount of memory reserved for send buffers for TCP socket. Each TCP socket has rights to use it due to fact of its birth. The default value is 4096 bytes.
  - Default - amount of memory allowed for send buffers for TCP socket by default. This value overrides wmem_default used by other protocols, it is usually lower than wmem_default. The default value is 16384 bytes.
  - Max - maximal amount of memory allowed for automatically selected send buffers for TCP socket. This value does not override wmem_max, "static" selection via SO_SNDBUF does not use this. The default value is 4194304 bytes.
- **Ipv4.tcp_mem**. Vector of 3 integers: min, pressure, max.
  - Low - below this number of pages TCP is not bothered about its memory appetite.
  - Pressure - when amount of memory allocated by TCP exceeds this number of pages, TCP moderates its memory consumption and enters memory pressure mode, which is exited when memory consumption falls under "low".
  - High - number of pages allowed for queuing by all TCP sockets.
    - Defaults are calculated at boot time from amount of available memory.
- **Tcp_timestamps**. Enable timestamps as defined in RFC1323. Enabled (1) by default.
- **Tcp_dsack**. Allows TCP to send "duplicate" SACKs. Enabled (1) by default.

- **Tcp_sack**. Enable Selective ACKnowledgement (SACK) Option for TCP. SACKs (RFC 2018) allow a receiver to acknowledge non-consecutive data. Enabled (1) by default.
- **Tcp_window_scalling**. Enable window scaling as defined in RFC1323. Enabled (1) by default. Is an option to increase the TCP receiving window size above its maximum value of 65,535 bytes.
- **Ip_forward**. Forward packets between interfaces if enabled (1). Disabled (0) by default. This variable is special; its change resets all configuration parameters to their default state (RFC1122 for hosts, RFC1812 for routers).
- **Tcp_fin_timeout**. Time to hold socket in state FIN-WAIT-2, if it was closed by our side. Peer can be broken and never close its side or even die unexpectedly. The default value is 60 seconds. Usual value used in 2.2 was 180 seconds you may restore it, but remember that if your machine is even underloaded web server, you risk to overflow memory with lots of dead sockets. FIN-WAIT-2 sockets are less dangerous than FIN-WAIT-1, because they eat maximum 1.5 kilobytes of memory, but they tend to live longer.
- **Tcp_keepalive_time**. How often TCP sends out keep-alive messages when keep-alive is enabled. The default value is 7200 seconds (2 hours).

A.1.3 Testing tools

There are different testing tools for benchmarking of web servers in the market. Here are two of the used during this report. It was chosen these two tools because they offered very good performance, and also good portability between different machines. Also, they were chosen because they worked in command line mode, without needing a graphical interface, which was not possible to use.

A.1.3.1 Apache Benchmark

Apache Benchmark tool is provided by the Apache foundation. It is a very powerful tool, letting the user configure the number of requests to perform and number of concurrent clients. Also, it allows saving the results of the tests in a gnuplot-file, as well as setting some POST information in the requests. Its working principle is very easy. There is no need to install the software as it comes with the Apache httpd installation (currently is installed with almost all Linux distributions). To launch the program is only need to be in the Apache folder and run **ab –n X –c Y [http://]hostname[:port]/path**.

Once the test is done, it is displayed in the screen some information about it. In first place it shows server information as for instance, the software version, hostname, and the server port. In second place it is the document information with the path and length of it. In third place there are some statistics of the connection, like the number of successful requests, total transferred bytes, requests per second and also response times. This information is very useful in order to view the performance of the web server in the test. Finally, it is displayed a table with some connection times and also the percentage of the requests served within a certain time.

A.1.3.2 Tsung

Tsung is a benchmark tool written in Erlang and can be distributed to use more than one machine to perform the test. Erlang is a programming language designed for building highly parallel, distributed, fault-tolerant systems. It has been used commercially for many years to build massive fault-tolerant systems, which run for years with minimal failures. Erlang combines ideas from the world of functional programming with techniques for building fault-tolerant systems to make a powerful language for building the massively parallel networked applications of the future.

To start the program, go to the folder where it is being installed and run it with **tsung -f myconfigfile.xml start** (by default is installed in /usr/lib/tsung). This will start the test that is provided in the myconfigfile.xml. After the test is finished, the results will be stored in the folder **~/.tsung/tsung_recorderYYYMMDD-HH:MM.xml**. There it will be found all the information of the performed test. To get the graphical results, it is needed to execute **/usr/lib/tsung/bin/tsung_stats.pl** file in the same folder where are stored all the results.

A.1.4 Apache httpd configuration

```
ServerRoot "/usr/local/apache2"
Listen 80
LoadModule ssl_module modules/mod_ssl.so
LoadModule uwsgi_module modules/mod_uwsgi.so
ScriptAlias /uwsgi/ "/usr/local/apache2/uwsgi/"
<Location /uwsgi>
        SetHandler uwsgi-handler
        uWSGISocket /tmp/uwsgi.sock
</Location>
<IfModule !mpm_netware_module>
<IfModule !mpm_winnt_module>
User apache
Group apache
</IfModule>
</IfModule>
ServerAdmin you@example.com
DocumentRoot "/usr/local/apache2/htdocs"
<Directory />
    Options FollowSymLinks
    AllowOverride None
    Order deny,allow
    Deny from all
</Directory>
<Directory "/usr/local/apache2/htdocs">
    Options Indexes FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
<IfModule dir_module>
    DirectoryIndex index.html
</IfModule>
<FilesMatch "^\.ht">
    Order allow,deny
    Deny from all
```

```
  Satisfy All
</FilesMatch>
ErrorLog "logs/error_log"
LogLevel warn
<IfModule log_config_module>
  LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined
  LogFormat "%h %l %u %t \"%r\" %>s %b" common
  <IfModule logio_module>
    LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\" %I %O" combinedio
  </IfModule>
  CustomLog "logs/access_log" common
</IfModule>
<IfModule alias_module>
  ScriptAlias /cgi-bin/ "/usr/local/apache2/cgi-bin/"
</IfModule>
<IfModule cgid_module>
</IfModule>
<Directory "/usr/local/apache2/cgi-bin">
  AllowOverride None
  Options None
  Order allow,deny
  Allow from all
</Directory>
DefaultType text/plain
<IfModule mime_module>
  TypesConfig conf/mime.types
  AddType application/x-compress .Z
  AddType application/x-gzip .gz .tgz
</IfModule>
# Various default settings:
Include conf/extra/httpd-default.conf
# Server-pool management (MPM specific):
Include conf/extra/httpd-mpm.conf
# PHP module configuration file:
Include conf/extra/php.conf
# Fast CGI module (mod_fcgid) configuration file:
Include conf/extra/fcgid.conf
# Secure (SSL/TLS) connections:
Include conf/extra/httpd-ssl.conf
<IfModule ssl_module>
SSLRandomSeed startup builtin
SSLRandomSeed connect builtin
</IfModule>
```

A.1.5 Lighttpd configuration

```
server.modules           = (
                         "mod_access",
                         "mod_fastcgi",
                         "mod_cgi",
                         "mod_accesslog" )
server.document-root     = "/usr/local/lighttpd/htdocs"
server.errorlog          = "/usr/local/lighttpd/logs/error.log"
index-file.names         = ( "index.php", "index.html",
                         "index.htm", "default.htm" )
server.tag               = "lighttpd/1.4.28 (UNIX)"
url.access-deny          = ( "~", ".inc" )
$HTTP["url"] =~ "\.pdf$" {
  server.range-requests = "disable"
}
static-file.exclude-extensions = ( ".php", ".pl", ".fcgi" )
server.pid-file          = "/var/run/lighttpd.pid"
server.max-worker = 4
server.max-fds = 65535
server.max-keep-alive-requests = 2000
server.max-keep-alive-idle = 90
server.max-read-idle = 60
server.max-write-idle = 360
server.event-handler = "linux-sysepoll"
server.network-backend = "linux-sendfile"
server.stat-cache-engine = "simple"
server.username          = "lighttpd"
server.groupname          = "lighttpd"
fastcgi.server            = ( ".php" =>
                         ( "tas01" =>
                          (
                           "socket" => "/tmp/php.sock",
                           "bin-path" => "/usr/bin/php-cgi",
                                                  "max-procs" => 16,
                                                  "bin-environment" => (
                                                      "PHP_FCGI_CHILDREN" => "12",
                                                      "PHP_FCGI_MAX_REQUESTS" => "10000" ),
                                            "bin-copy-environment" => ("PATH", "SHELL", "USER" )
                          )),
                                        ".py" =>
                                               ( "tas01" =>
                                                (
                                                  "socket" => "/tmp/py-fcgi.socket",
                                                  "bin-path" => "/usr/local/lighttpd/htdocs/pytestfcgi.py"
                                                 ))
                        )
#cgi.assign               = ( ".pl"  => "/usr/bin/perl",
#                          ".cgi" => "/usr/bin/perl",
#                                               ".php" => "/usr/bin/php-cgi",
#                                               ".py" => "/usr/bin/python" )
#
#ssl.engine               = "enable"
#ssl.pemfile              = "/usr/local/lighttpd/sbin/host.pem"
#ssl.use-sslv2             = "disable"
#ssl.cipher-list          = "HIGH:MEDIUM:!ADH"
```

A.1.6 Nginx configuration

```
user  devel;
worker_processes  2;
worker_rlimit_nofile 65535;
error_log  logs/error.log;
events {
   worker_connections  16384;
   use epoll;
}
http {
   include       mime.types;
   default_type  application/octet-stream;
   sendfile       on;
   keepalive_timeout  90;
   server {
      listen       80;
      server_name  tas01;
      location / {
         root   html;
         index  index.html index.htm index.php;
      }
      error_page   500 502 503 504  /50x.html;
      location = /50x.html {
         root   html;
      }
      location ~ \.php$ {
         fastcgi_pass   127.0.0.1:3000;
               include       fastcgi_params;
      }
            location ~ \.py$ {
         include uwsgi_params;
               uwsgi_pass unix:/tmp/uwsgi.sock;
      }
   }
   #ssl_session_cache    shared:SSL:10m;
   #ssl_session_timeout  10m;
   #server {
   #    listen       443;
   #    server_name  tas01;
   #    ssl                          on;
   #    ssl_certificate              /usr/local/nginx/conf/host.pem;
   #    ssl_certificate_key        /usr/local/nginx/conf/tas01.key;
   #    ssl_session_timeout     70;
   #    ssl_protocols                          SSLv3 TLSv1;
   #ssl_ciphers ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP;
#       ssl_prefer_server_ciphers   on;
#       location / {
#           root   html;
#           index  index.html index.htm;
#       }
#           location ~ \.php$ {
#           root           html;
#               fastcgi_pass   127.0.0.1:3000;
#           fastcgi_index  index.php;
#           include        fastcgi_params;
#       }
#       location ~ \.py$ {
#               root html;
#           include uwsgi_params;
#           uwsgi_pass unix:/tmp/uwsgi.sock;
#       }
#   }
}
```

A.1.7 Cherokee configuration

```
config!version = 001002000
server!bind!1!port = 80
server!bind!1!tls = 0
server!chunked_encoding = 1
server!fdlimit = 65535
server!iocache = 0
server!ipv6 = 1
server!keepalive = 1
server!keepalive_max_requests = 2000
server!max_connection_reuse = 1000
server!panic_action = /usr/local/cherokee/bin/cherokee-panic
server!pid_file = /usr/local/cherokee/stat/run/cherokee.pid
server!poll_method = epoll
server!server_tokens = full
server!thread_policy = other
server!timeout = 90
vserver!1!directory_index = index.html, index.php
vserver!1!document_root = /usr/local/cherokee/htdocs
vserver!1!error_writer!filename = /usr/local/cherokee/stat/log/cherokee.error
vserver!1!error_writer!type = file
vserver!1!nick = default
vserver!1!rule!1200!disabled = 0
vserver!1!rule!1200!document_root = /usr/local/cherokee/wordpress
vserver!1!rule!1200!encoder!deflate = forbid
vserver!1!rule!1200!encoder!gzip = forbid
vserver!1!rule!1200!match = directory
vserver!1!rule!1200!match!directory = /wordpress
vserver!1!rule!1200!match!final = 0
vserver!1!rule!1100!disabled = 0
vserver!1!rule!1100!handler = fcgi
vserver!1!rule!1100!handler!balancer = round_robin
vserver!1!rule!1100!handler!balancer!source!10 = 1
vserver!1!rule!1100!match = extensions
vserver!1!rule!1100!match!check_local_file = 0
vserver!1!rule!1100!match!extensions = php
vserver!1!rule!1100!match!final = 0
vserver!1!rule!900!disabled = 1
vserver!1!rule!900!document_root = /usr/local/cherokee/php_fpm
vserver!1!rule!900!encoder!deflate = forbid
vserver!1!rule!900!encoder!gzip = forbid
vserver!1!rule!900!handler = fcgi
vserver!1!rule!900!handler!balancer = round_robin
vserver!1!rule!900!handler!balancer!source!10 = 5
vserver!1!rule!900!match = directory
vserver!1!rule!900!match!directory = /php_fpm
vserver!1!rule!900!match!final = 1
vserver!1!rule!800!disabled = 1
vserver!1!rule!800!document_root = /usr/local/cherokee/scgi-bin
```

```
vserver!1!rule!800!handler = scgi
vserver!1!rule!800!handler!balancer = round_robin
vserver!1!rule!800!handler!balancer!source!10 = 3
vserver!1!rule!800!handler!check_file = 0
vserver!1!rule!800!handler!error_handler = 1
vserver!1!rule!800!handler!pass_req_headers = 1
vserver!1!rule!800!handler!xsendfile = 0
vserver!1!rule!800!match = directory
vserver!1!rule!800!match!directory = /scgi-bin
vserver!1!rule!700!disabled = 1
vserver!1!rule!700!document_root = /usr/local/cherokee/scgi-py
vserver!1!rule!700!encoder!deflate = forbid
vserver!1!rule!700!encoder!gzip = forbid
vserver!1!rule!700!handler = uwsgi
vserver!1!rule!700!handler!balancer = round_robin
vserver!1!rule!700!handler!balancer!source!10 = 4
vserver!1!rule!700!match = directory
vserver!1!rule!700!match!directory = /scgi-py
vserver!1!rule!600!disabled = 1
vserver!1!rule!600!document_root = /usr/local/cherokee/fcgi-bin
vserver!1!rule!600!encoder!deflate = forbid
vserver!1!rule!600!encoder!gzip = forbid
vserver!1!rule!600!handler = fcgi
vserver!1!rule!600!handler!balancer = round_robin
vserver!1!rule!600!handler!balancer!source!10 = 1
vserver!1!rule!600!match = directory
vserver!1!rule!600!match!directory = /fcgi-bin
vserver!1!rule!500!encoder!gzip = allow
vserver!1!rule!500!handler = server_info
vserver!1!rule!500!handler!type = just_about
vserver!1!rule!500!match = directory
vserver!1!rule!500!match!directory = /about
vserver!1!rule!400!document_root = /usr/local/cherokee/cgi-bin
vserver!1!rule!400!handler = cgi
vserver!1!rule!400!match = directory
vserver!1!rule!400!match!directory = /cgi-bin
vserver!1!rule!300!document_root = /usr/local/cherokee/share/cherokee/themes
vserver!1!rule!300!handler = file
vserver!1!rule!300!match = directory
vserver!1!rule!300!match!directory = /cherokee_themes
vserver!1!rule!200!document_root = /usr/local/cherokee/share/cherokee/icons
vserver!1!rule!200!handler = file
vserver!1!rule!200!match = directory
vserver!1!rule!200!match!directory = /icons
vserver!1!rule!100!encoder!deflate = forbid
vserver!1!rule!100!encoder!gzip = forbid
vserver!1!rule!100!handler = common
vserver!1!rule!100!handler!iocache = 0
vserver!1!rule!100!match = default
source!1!env!PHP_FCGI_CHILDREN = 12
source!1!env!PHP_FCGI_MAX_REQUESTS = 5000
source!1!env_inherited = 0
source!1!host = /tmp/php.socket
source!1!interpreter = /usr/bin/php-cgi -b /tmp/php.socket
source!1!nick = php
source!1!type = interpreter
source!2!env_inherited = 0
source!2!host = /tmp/scgi-perl.socket
source!2!interpreter = /usr/bin/perl /usr/local/cherokee/scgi-bin/perltest.pl
source!2!nick = perl
source!2!type = interpreter
source!3!env_inherited = 1
source!3!host = 127.0.0.1:3040
source!3!interpreter = /usr/local/cherokee/scgi-bin/start_fcgi.sh
source!3!nick = python
source!3!type = interpreter
source!4!env_inherited = 1
source!4!host = /tmp/uwsgi.sock
source!4!interpreter = /usr/local/bin/uwsgi/uwsgi -s /tmp/uwsgi.sock -M -w /usr/local/cherokee/scgi-py/t -p 4
source!4!nick = uwsgi
source!4!type = interpreter
source!5!env_inherited = 1
source!5!host = 127.0.0.1:3000
source!5!interpreter = /etc/init.d/php-fpm start
source!5!nick = php-fpm
source!5!type = host
```

A.1.8 Yaws configuration

```
logdir = /usr/local/yaws/var/log/yaws
ebin_dir = /usr/local/yaws/lib/yaws/examples/ebin
ebin_dir = /usr/local/yaws/var/yaws/ebin
include_dir = /usr/local/yaws/lib/yaws/examples/include
max_connections = nolimit
# collecting too much garbage in the erlang VM.
keepalive_maxuses = nolimit
process_options = "[]"
trace = false
use_old_ssl = false
copy_error_log = true
log_wrap_size = 1000000
log_resolve_hostname = false
fail_on_bind_err = true
auth_log = false
pick_first_virthost_on_nomatch = true
keepalive_timeout = 10
use_fdsrv = false
<server tas01>
        port = 8080
        listen = 0.0.0.0
                docroot = /usr/local/yaws/htdocs
                phpfcgi = 127.0.0.1:3000
                access_log = false
</server>
<server tas01>
        port = 8081
        listen = 0.0.0.0
        docroot = /usr/local/yaws/htdocs
        fcgi_app_server = 127.0.0.1:5000
                access_log = false
</server>
#<server tas01>
#       port = 80
#       listen = 0.0.0.0
#       docroot = /usr/local/yaws/htdocs
#       appmods = <cgi-bin, yaws_appmod_cgi>
#               access_log = false
#</server>
#<server tas01>
#       port = 443
#       docroot = /usr/local/yaws/htdocs
#       listen = 0.0.0.0
#       dir_listings = true
#               access_log = false
#       <ssl>
#               keyfile = /usr/local/yaws/etc/yaws/yaws-key.pem
#               certfile = /usr/local/yaws/etc/yaws/yaws-cert.pem
#                               keyfile = /usr/local/yaws/etc/yaws/tas01.key
#               certfile = /usr/local/yaws/etc/yaws/tas01.crt
#               depth = 0
#                               ciphers = AES256-SHA
#       </ssl>
#</server>
```

A.1.9 Mongrel2 configuration

```
main = Server(
    uuid="f400bf85-4538-4f7a-8908-67e313d515c2",
    access_log="/logs/access.log",
    error_log="/logs/error.log",
    chroot="./",
    default_host="tas01",
    name="tas01",
    pid_file="/run/mongrel2.pid",
    port=8080,
    hosts = [
        Host(name="tas01", routes={
            '/': Dir(base='htdocs/', index_file='index.html',
                        default_ctype='text/plain')
        })
    ]
)

servers = [main]
```

Add to the end of the configuration file. Does not work properly, but it could be the next feature of the web server, SSL support:

```
settings = {
  "f400bf85-4538-4f7a-8908-67e313d515c2.use_ssl": 1,
  "certdir": "./certs/"
}
```

A.1.10 Apache mod_PHP configuration

```
LoadModule php5_module modules/libphp5.so
<Location />
AddType text/html .php .phps
AddHandler application/x-httpd-php .php
AddHandler application/x-httpd-php-source .phps
</Location>
```

A.1.11 Apache FastCGI configuration

```
LoadModule fcgid_module modules/mod_fcgid.so
<IfModule mod_fcgid.c>
    AddHandler  fcgid-script .fcgi .fcg .fplq
    IdleTimeout 300
    ProcessLifeTime 3600
    MaxProcessCount 1000
    DefaultMinClassProcessCount 3
    DefaultMaxClassProcessCount 100
    IPCConnectTimeout 60
    IPCCommTimeout 60
    BusyTimeout 120
    SocketPath  /tmp/fcgi-ipc
    FCGIWrapper /usr/local/apache2/fcgi-bin/php5-fcgi .php
    AddType     application/x-httpd-php5    .php .php5
</IfModule>
ScriptAlias /fcgi-bin/ "/usr/local/apache2/fcgi-bin/"
<Location /fcgi-bin/>
    Options +ExecCGI
    SetHandler fcgid-script
    Order allow,deny
    Allow from all
</Location>
```

A.2 Test cases extended

Here is an extended list of all the test cases that have been configured and reproduced. As said back in the **Chapter 3** of this project, there are few considerations about them.

- All tests were run multiple times to assure repeatability.
- Performance was measured in the Web Server side (to know the CPU load, and RAM usage) and in the client side (to know the requests per second and the system's load).
- During the test, no other applications were running and using resources on the system under test.
- If something is changed or added to web server's configuration file, it will be explained during the test case.

There are five types of tests. The first one is static content, in which will be requested a static web page, a small image and a large image. This test will show how optimized is each web server in front of static content. The second test consists in requesting different dynamic languages, as for example PHP and Python. Those languages depend of an external interpreter, but the way the web server handles the requests will influence the final result.

The third test case is using secure communications between the client and the server, by means of HTTPS. It will be possible to see if there is any performance penalty by enabling such feature. During the entire tests explained before, it will be performed the same test twice, first without using keep-alive and next enabling it. Nowadays, almost all web browsers and applications make use of keep-alive functionality of the HTTP protocol, it is important to see the performance variation by using it.

The last test will be the load test, where it will be loaded a blog web site. This test will provide us with more real performance numbers, as it will be installed a commercial free blog tool, Wordpress. During this test case it will also be used some cache mechanisms, in order to get more performance, like in the real deployment environment.

A.2.1 Static tests

A.2.1.1 HTML

This benchmark consists in request an HTML file of 168 bytes. This file only includes few lines of code, where is shown only a test sentence, This a webserver test page. In the TAS04 machine it will be started the tool AB and perform the ST-1 HTML test with the different load conditions.
Ab –n X –c Y http://tas01/webtest.html.
Where, X and Y are the requests and concurrency loads respectively.

```
<!CTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
```

```
<head>
<title>Webserver test</title>
</head>
<body>
This is a webserver test page.
</body>
</html>
```

Here it will be expected the network not to be the main problem, rather than the capacity of the web server of getting profit of the processor or getting the processor extremely loaded.

$$\frac{hit}{s} = \frac{940\ x\ 10^6\ bit/_{1\ s}}{168\ Bytes * 8\ bits/_{1\ Byte}} = 699.404\ hit/_{s}$$

A.2.1.2 Small image

This benchmark consists in request a small image file of 7,500 bytes with a dimension of 203x61 pixels. This image is a small one, but is about 40 times larger than the previous test. In TAS04 machine it will be started the tool AB and perform the ST-2 Image small test with the different load conditions.
Ab –n X –c Y http://tas01/image_small.gif.
Where, X and Y are the requests and concurrency loads respectively.

Here it is expected the network to be a possible bottleneck. As seen before, the maximum bandwidth of the network is about 940Mbps, so doing some calculations it is going to get the maximum rate in requests/s of the test.

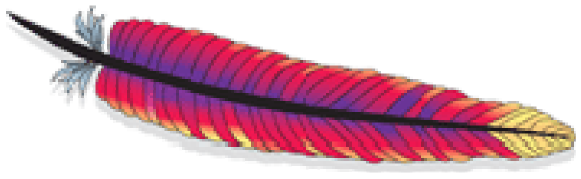

Fig. A.1: ST-2-Image small test

$$\frac{hit}{s} = \frac{940\ x\ 10^6\ bits/_{1\ s}}{7500\ Bytes * 8\ bits/_{Byte}} = 15.666\ hit/_{s}$$

So it is not possible to expect getting more requests per second that those that have been calculated. Also as the files are larger than the MTU (set to 1500 bytes), will be overhead the result will be slightly smaller than the maximum that have been calculated previously.

A.2.1.3 Large image

This benchmark consists in request a larger file with a size of 83,572 bytes with a dimension of 1600x1200 pixels. This image is 10 times larger than the

previous one. Here it is expected the network to be the main problem. As seen before, the maximum bandwidth of the network is about 940Mbps, so doing some calculations it will generate the maximum rate in requests/s of the test. In TAS04 machine it will be started the tool AB and perform the ST-3 Image large test with the different load conditions.
Ab –n X –c Y http://tas01/image_large.jpeg.
Where, X and Y are the requests and concurrency loads respectively.

Fig. A.2: ST-3-Image large test

$$\frac{hit}{s} = \frac{940\ x\ 10^6\ {bits}/{1\ s}}{83572\ Bytes * 8\ {bits}/{Byte}} = 1.405\ {hit}/{s}$$

So it is not possible to expect getting more requests per second that those that have been calculated. Also as the files are larger than the MTU (set to 1500 bytes), will be overhead the result will be slightly smaller than the maximum that have been calculated previously.

A.2.2 Dynamic tests

A.2.2.1 PHP

This benchmark consists in requesting a dynamic page written in PHP. The page will be a PHP file of 14 bytes, only returning a Hello, World! sentence. As there are different types of web servers it is defined three different types of tests according to the possibilities of each one of them. There is no need to calculate the maximum number of requests that the network can handle, due to the fact that processing a dynamic language will load the CPU of the machine, limiting the requests per second. Although being a very small file, performance will depend on how it is executed in the web server side (module/CGI/Fast CGI).

```
<?php
   echo "Hello, World!";
?>
```

If it is calculated the maximum number of requests per second, it will be a higher number than before, but it will be never reached. As there is the need to interpret the language, performance of the web server will decrease. The expected performance is much lower, due to the cost of process the dynamic language.

$$\frac{hit}{s} = \frac{940\ x\ 10^6\ bits/_{1\ s}}{14\ Bytes * 8\ bits/_{Byte}} = 8.390.857\ hit/_s$$

A.2.2.2 Python

This benchmark consists in requesting a dynamic page written in Python. The page will be a Python file of bytes 750 bytes. As there are different types of web servers it is defined three different types of tests according to the possibilities of each one of them. If it is calculated the maximum number of requests per second, it will get a lower number than before, but, nevertheless it will be never reached. As there is the need to interpret the language, performance of the web server will decrease. The expected performance is much lower, due to the cost of process the dynamic language.

$$\frac{hit}{s} = \frac{940\ x\ 10^6\ bits/_{1\ s}}{750\ Bytes * 8\ bits/_{Byte}} = 156.666\ hit/_s$$

- **CGI execution:**

```
#!/usr/bin/python
print "Content-type: text/html"
print
text = "This is a test;"
text2 = text*50
print text2
```

- **FastCGI execution:**

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
from flup.server.fcgi import WSGIServer

def app(environ, start_response):
   start_response('200 OK', [('Content-Type', 'text/html')])
   text = "This is a test;"
   text2 = text*50
   yield text2
WSGIServer(app).run()
```

- **uWSGI execution:**

```
def application(environ, start_response):
   start_response('200 OK', [('Content-Type', 'text/plain')])
   text = "This is a test;"
   text2 = text*50
   yield text2
```

For uWSGI test it is needed to change the structure of the script a little bit, as it only needs to define an application to be executed. The structure of the application is the same than for Fast CGI, but without includes and the execution line. To run uWSGI processes in the machine, it must be installed the uWSGI from the author's web page and spawn the processes. If it is installed uWSGI in the */usr/local/bin/uwsgi* it will be started the process this way: **/usr/local/bin/uwsgi/uwsgi/ -s /tmp/uwsgi.sock –M –w /usr/local/webserver/uwsgi-bin/pythontest –L –p4**.

This will call the uWSGI program, bind it to the UNIX socket */tmp/uwsgi.sock*, creating a master process and finally indicating which file to execute (without the extension of the file, .py). Also it is possible to define, as in PHP, how many uWSGI processes to spawn, in this case 4.

A.2.2.3 SERVLET

Tomcat offers the possibility of running Servlets in the core of the web server. Here it will be benchmarked the Tomcat Servlet feature. Performance of this test is expected to be higher than for PHP test case. The script consists in showing in the screen the headers of request. It is about 670 bytes.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestHeaderExample extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    Enumeration e = request.getHeaderNames();
    while (e.hasMoreElements()) {
      String name = (String)e.nextElement();
      String value = request.getHeader(name);
      out.println(name + " = " + value);
    }
  }
}
```

In TAS04 machine it will be started the tool AB and perform the Servlet test with the different load conditions.
Ab –n X –c Y **http://tas01/examples/servlets/servlet/RequestHeaderExample**.
Where, X and Y are the requests and concurrency loads respectively.

A.2.2.4 JSP

Tomcat offers JSP compatibility. It is expected that the performance of this test will be higher to the Servlet one. The script consists in showing calendar information. It is size of 373 bytes.

```
<html><%@ page session="false"%><body bgcolor="white">
<jsp:useBean id='clock' scope='page' class='dates.JspCalendar' type="dates.JspCalendar" />
<font size=4><ul>
```

```
<li>Day of month: is  <jsp:getProperty name="clock" property="dayOfMonth"/>
<li>Year: is  <jsp:getProperty name="clock" property="year"/>
<li>Month: is  <jsp:getProperty name="clock" property="month"/>
<li>Time: is  <jsp:getProperty name="clock" property="time"/>
<li>Date: is  <jsp:getProperty name="clock" property="date"/>
<li>Day: is  <jsp:getProperty name="clock" property="day"/>
<li>Day Of Year: is  <jsp:getProperty name="clock" property="dayOfYear"/>
<li>Week Of Year: is  <jsp:getProperty name="clock" property="weekOfYear"/>
<li>era: is  <jsp:getProperty name="clock" property="era"/>
<li>DST Offset: is  <jsp:getProperty name="clock" property="DSTOffset"/>
<li>Zone Offset: is  <jsp:getProperty name="clock" property="zoneOffset"/>
</ul></font></body></html>
```

In TAS04 machine it will be started the tool AB and perform the Servlet test with the different load conditions.
Ab –n X –c Y **http://tas01/examples/jsp/dates/date.jsp**.
Where, X and Y are the requests and concurrency loads respectively.

A.2.3 HTTPS

It will be tested some static and dynamic content in HTTPS mode to see how it performs against not secure protocol. To be able to perform those tests, it will be configured the web servers to support secure connections. For this reason it will be generated some certificates, in order to enable the SSL feature in the web server.

For generating all the certificates it was used OpenSSL application. Next are the followed steps in order to create the needed certificates. All the steps are commented to explain what is being done. In first place, it is needed to generate a private key. There are different possibilities, for instance using Triple-DES, the most secure one, or the RSA key. In this case it will be used the RSA, as it is not needed the extra security which provides the Triple-DES, since this is only an example, not a production web server.

```
openssl genrsa > tas01.key
```

The second step is to generate the CSR (Certificate Signing Request). Now, it is necessary to generate the request to send to a CA (Certificate Authority) and request to sign the key and return one certificate.

```
Openssl req –new –key tas01.key > tas01.csr
```

It is needed to answer some questions about, like country, city, company, name and email. Afterwards, this certificate could be send to a CA like Verisign, and after paying the cost of certificating it will get our signed certificate. But, this is not the main purpose of this report, so, it will be created our CA. It is needed to locate the file CA.pl and create a new CA.

```
/etc/pki/tls/misc/CA.pl –newca
```

Next, it is needed to create a new file where it will be loaded the CA info. It must be entered a password, the country name, state, locality, organization name, organization unit, server host name and email address. Now, it has been created a new CA. Executing the next command will create the certificates signed by the CA.

```
sudo openssl ca -policy policy_anything -out tas01.crt -infiles tas01.csr
```

To create a self-signed certificate it is needed to do the following:

```
openssl x509 -req -days 3650 -in tas01.csr -signkey tas01.key -out newcert.pem
```

Finally, it is needed to move the certificate files to any web server folder. For the Apache httpd it will be placed the files inside the configuration folder to provide the web server of the tas01.crt and tas01.key.

```
Cp tas01.key /usr/local/apache2/conf
Cp tas01.crt /usr/local/apache2/conf
```

Now, it is needed to edit the Apache's configuration file in order to accept all the SSL communications and with the certificates that have been created. For this reason it will be included the *http-ssl.conf* file which is inside the */conf/extra* folder (inside the Apache httpd installation folder). In the *httpd.conf* file it is needed to add the following lines (or maybe only uncomment it):

```
LoadModule ssl_module modules/mod_ssl.so
Include conf/extra/httpd-ssl.conf
```

And in http-ssl.conf it is only needed to change the following lines:

```
SSLCertificateFile "/usr/local/apache2/conf/tas01.crt"
SSLCertificateKeyFile "/usr/local/apache2/conf/tas01.key"
```

It can be started the web server and point the browser to the https://tas01/webtest.html and accept the certificate that has been created minutes ago. To test the web server with **ab** benchmark tool, it must be compiled with SSL support.

A.2.4 Load tests

This test will be performed by using a distributed load testing tool, called Tsung. Tsung is based on Erlang language, which makes it a perfect distributive benchmarking tool, taking advantage of Erlang's OTP libraries. It is needed to set up a XML-like script, where it will be set the different involving machines, as well as the benchmarking parameters.

```
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/usr/share/tsung/tsung-1.0.dtd" []>
<tsung loglevel="warning">
  <clients>
    <client host="tas04" use_controller_vm="false" maxusers="2000"/>
    <client host="tas02" use_controller_vm="false" maxusers="2000"/>
    <client host="tas03" use_controller_vm="false" maxusers="2000"/>
  </clients>
  <servers>
    <server host="tas01" port="80" type="tcp"/>
  </servers>
  <load>
    <arrivalphase phase="1" duration="30" unit="minute">
```

```
      <users arrivalrate="100" unit="second"/>
    </arrivalphase>
    <arrivalphase phase="2" duration="10" unit="minute">
      <users arrivalrate="450" unit="second"/>
    </arrivalphase>
    <arrivalphase phase="3" duration="20" unit="minute">
      <users arrivalrate="200" unit="second"/>
    </arrivalphase>
   </load>
  <sessions>
    <session name='Nginx' probability='100' type='ts_http'>
        <request>
          <http url='http://tas01/wordpress/' version='1.1' method='GET'/>
        </request>
        <thinktime random='false' value='5'/>
        <request>
          <http url='http://tas01/wordpress/?p=1' version='1.1' method='GET'/>
        </request>
    </session>
  </sessions>
</tsung>
```

A.3 Test results extended

Not all the obtained results could be presented during the main part of this report. So, in this appendix it will be given the rest of the results, which could give a widely vision of the different tested web servers.

A.3.1 Static tests

As said during the report, the static tests consist in three different scenarios, where it will be requested an HTML file, a small image and a large image. Those tests are not computational expensive for the web server as there are not any dynamic language to process, but if generating so much concurrent clients the server will be loaded. Eventually, it will be the CPU and the RAM memory which will limit the overall performance.

Resources are a limited aspect in any computer, and the operating system assigns those available resources to any process that needs it. CPU load is a limitation as will mark how quickly processes do their schedule job. Meanwhile, RAM memory is a quick access memory, where process access to exchange data in a quicker way. This RAM memory is also a finite resource, and although it is released each time a process ends its schedule job, it may be not as quick as the operating system needs. If the operating system runs out of RAM memory it will start to save data into the hard disk, which will slow down the whole system.

Table A.8: Web server memory footprint

	Worker		**Master**	
	VmRSS (kB)	**VmSize (kB)**	**VmRSS (kB)**	**VmSize (kB)**
Apache Prefork	2156	69416	2156	69416
Apache Worker	2176	380872	35036	102180
Apache Event	2160	364356	18496	85664
Lighttpd	716	48308	716	48308
Nginx	7196	47240	896	41084
Cherokee	4852	142792	752	25868
Mongrel 2	8264	151912	1668	22760
Yaws	41504	171976	-	-
Tomcat	79912	1338188	-	-

Table A.8 shows the memory footprint of all used web servers during this project. It has four different columns, which are the next:

- **Worker**. It is the active process which answers the incoming requests.
- **Master**. It is the main process of the web server, which will work as a middleware in order to distribute the load between the different worker processes.

- **VmRSS**. Size of memory resident set currently in physical memory including Code, Data, and Stack. It explains how many of the allocated blocks owned by the task currently reside in RAM.
- **VmSize**. Virtual memory usage of entire process

Summing up this memory footprint introduction, it is important to highlight that the less consumption of memory the better. This will let more processes being created and the less probability of running out of memory. There is only one master process for each web server, which controls the workers processes and spawns more as it is specified in the configuration. Consumption of RAM memory depends in each web server the configuration.

Apache httpd web server it is presented with its different configurations. Prefork mode works as a multi process single thread configuration, which has the same memory allocated for the worker and for the master process up to 2MB of RAM and up to 69MB of assigned virtual memory. For Apache's Worker, the consumption of RAM memory is almost the same than the previous, 2MB, but the assigned virtual memory is larger, with up to 380MB. Apache's master process of worker configuration has a larger memory footprint, up to 35MB, but lower virtual memory assigned, 102MB.

For Apache's Event configuration, the assigned memory is more or less the same than for Worker's configuration, although its master process assigned memory is smaller. If it is compared Apache's memory footprint with other web servers, it is possible to see that it has higher memory consumption than others, something that could be a negative impact in the system's performance.

Lighttpd memory footprint is very low, only having allocated 716KB of RAM and a total assigned memory of 48MB, which is the same for the worker and for the master process. Increasing the number of worker processes raises the memory consumption, but few KB. The other event-driven web server, Nginx, does present higher RAM consumption for its worker process, 7MB, although it has a low total virtual memory assigned, 47MB. Nginx's master process consumes as low as Lighttpd.

Cherokee web server memory footprint is also low, with only 4.8MB and a total virtual memory of 142MB for its worker process. Meanwhile, for the master process, the memory consumption is lower, with only 752KB and 26MB respectively. Cherokee only has one active worker process, so it will not consume so much memory for static tasks. Tomcat is the other multithread web server. Tomcat memory footprint is the highest one found in this report, with a total of 80MB for its worker process and a total of 1,338MB of assigned virtual memory. It does not have a master process and it is a single process web server, but nevertheless its RAM consumption is high.

Mongrel2 memory footprint is like Nginx's, 4.8MB, although its maximum allocated virtual memory is higher, 142MB. Yaws has a large memory footprint with up to 41MB per process.

A.3.1.1 HTML

During the report it has been said that event-driven web servers do offer high scalability when using different worker configurations. Such scalability factor it is reached when using different active worker configurations, as well as using keep-alive feature of HTTP protocol. Here, are shown the differences by using or not keep-alive.

The first event-driven web server to test is Lighttpd. As seen in **Table A.3**, its memory footprint is very low, which does not affect the overall performance of the system. Also, it is tested with different worker configurations to see its performance. **Fig A.3** shows the evolution of the performance of Lighttpd when not using keep-alive. It is possible to see that the default configuration, with one active worker, does offer almost the same performance as the other configurations, multi worker approach.

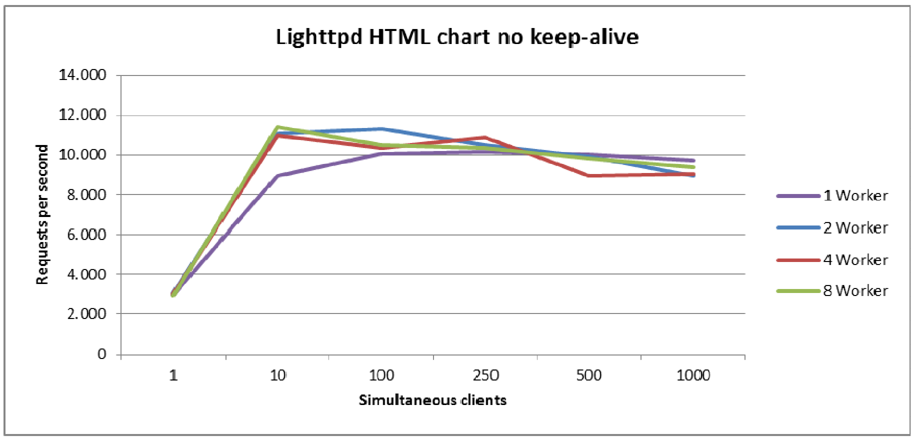

Fig. A.3: Lighttpd HTML no keep-alive chart

Also, it is possible to see that there is not much gain by increasing the number of active workers as they perform very similar. Although, Lighttpd official documentation advices to use 4 worker configuration in case of wanting an increase of performance. Here, in this case, it may be not the best configuration compared with the other, but it performs better than with only one active worker. Using keep-alive, performance results changes a lot (see **section 4.1.1**), performance increases, reaching up to 30,000 requests per second, which is more than 150%.

The other event-driven web server is Nginx. It shares the same architecture than Lighttpd, although it may handle requests in a different way. **Fig. A.4** shows the performance of Nginx's multi process configurations with no keep-alive mode. It is possible to see, like in Lighttpd's case, that performance is limited, no matter how many active workers are set up. Contrary to Lighttpd, Nginx's performance decreases as increasing the number of simultaneous clients. Nginx's official documentation advices to use the same number of active workers as the number of processors that has the system, in this case two. It is possible to see that there is not so much difference regarding the configuration.

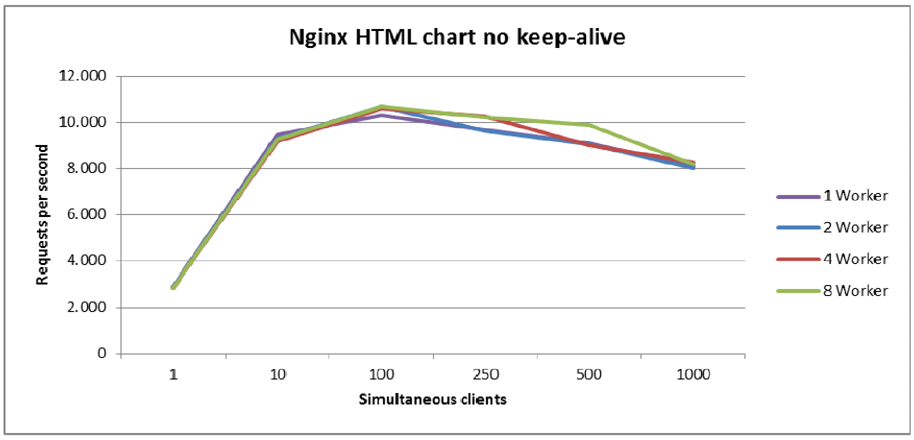

Fig. A.4: Nginx HTML no keep-alive chart

To end this test case results round, it is provided with a chart of the best performing web servers. In this chart, **Fig. A.5**, it is shown the best configurations of each web server.

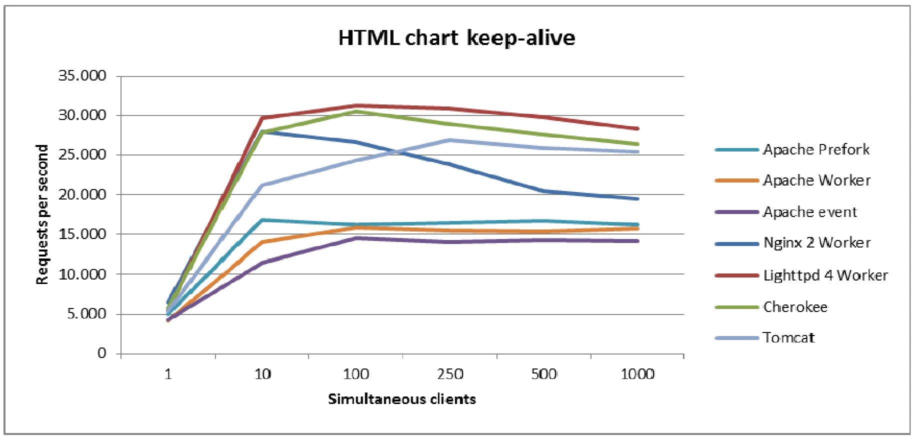

Fig. A.5: Best performing web server in HTML keep-alive test

Although Apache web server offers a very good and stable performance, it is possible to see that it tops at 15,000 requests per second, far away from other solutions. The three configurations of Apache have a very similar performance, being better the prefork, which is the default installation (see **Fig. A.5**).

Event-driven web servers have a very good performance. Lighttpd reaches a top performance of 31,000 requests per second, which is the best result of the test, keeping this rate during the entire test. On the other hand, Nginx has a good performance, reaching up to 27,000 requests per second, but decreases when increasing the number of simultaneous clients (see **Fig. A.5**).

Multithread web servers have also very good performance. Cherokee performs very similar to Lighttpd, using both very different architecture approaches. It tops at 30,000 requests per second, decreasing when increasing the number of simultaneous clients. Furthermore, Tomcat, which is an application server, has a very good performance when serving static content. It increases its performance as raising the number of simultaneous clients, exceeding Nginx's performance. The top performance of Tomcat is around 26,000 requests per second.

To conclude this section it is possible to say that the best web server for this test is Lighttpd, which outperforms the rest of tested web servers, reaching up to 31,000 requests per second. The second best web server is the single process multithread Cherokee, which reaches up to 30,000 requests per second. Apache, which is the king of the web servers, it is not the best one in this test case, because it is not optimized for static content rather than for dynamic content.

A.3.1.2 Small image

The small image test was limited by the network bandwidth, as it was calculated in previous sections of the appendix. As in HTML test case, event-driven web servers are expected to gain some performance by enabling its multi process configuration. Lighttpd performance (see **Fig. A.6**) is limited, getting up to 8,000 – 9,000 requests per second. Multi process configurations are better than default, although it does not represent any special gain.

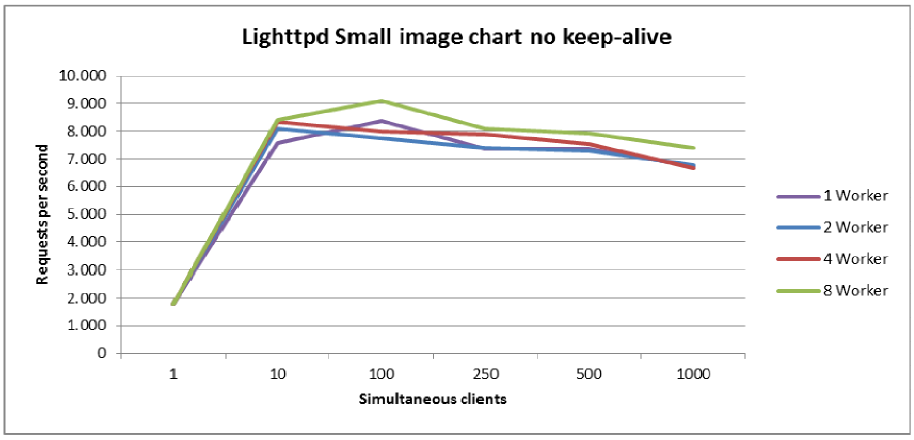

Fig. A.6: Lighttpd small image no keep-alive chart

Enabling keep-alive does represent a high performance increase (see **section 4.1.2**). The increase by using keep-alive in front of not using it is about 75%.

Nginx's case is very similar than the previous one. Performance achieved without using keep-alive feature is very limited, up to 8,000 requests per second (see **Fig. A.7**). Multi process configuration does not help the web server to increase its performance. As happened in the previous test case, the HTML, performance of Nginx decreases as the number of simultaneous clients increase.

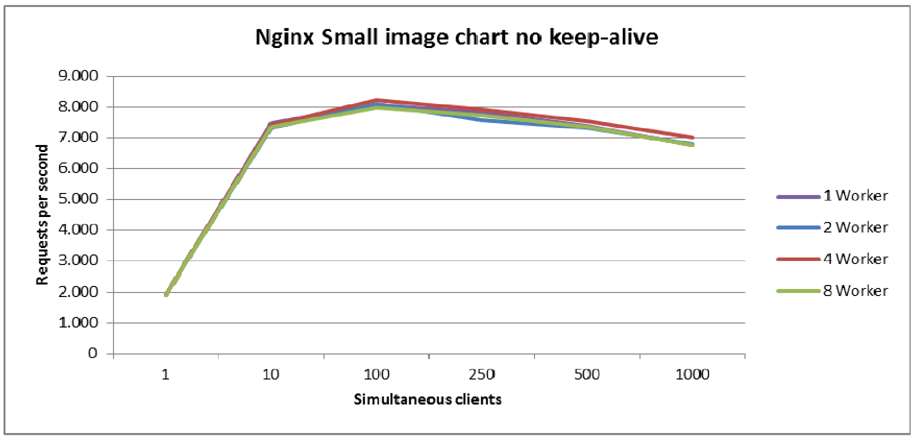

Fig. A.7: Nginx small image no keep-alive chart

Using keep-alive increases the performance of the web server (see **section 4.2.1**), like in Lighttpd's case. Although, the use of multi process configuration does not increase so much the obtained performance. Nevertheless, the performance of this test is limited by the network bandwidth not by the CPU or RAM memory.

The latest chart (see **Fig. A.8**) to show is meant to put together the best performing web servers of this test case. It is possible to see that the four web servers have the same performance, reaching the top rate at 15,000 requests per second. As simultaneous clients increase there is a stable response of all the web servers, although Nginx decreases it a little bit.

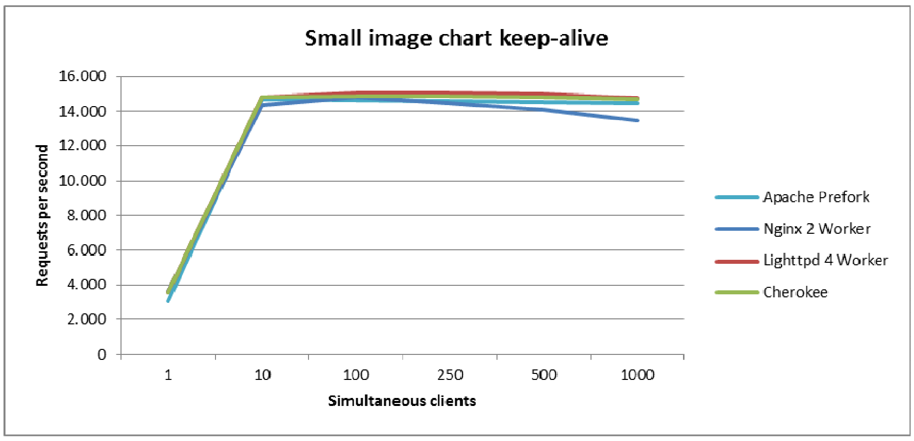

Fig. A.8: Best performing web server in small image test

A.3.2 Dynamic tests

Dynamic tests consist in requesting some web page which has some dynamic language embedded into. During this project are tested different dynamic languages, such as PHP, Python, and Java. As said before, processing dynamic web pages is more computational expensive than static pages, this is due to the fact that it is needed an external interpreter to process it, and afterwards serve the answer.

A.3.2.1 PHP

The first language to test is PHP. PHP is a widely used dynamic language in the Internet. It powers a lot of web pages and a large number of the most used frameworks, such as blogs, newspapers, media content, etc. There are lots of different connectors in order to link with different programs, like databases, other frameworks, etc. As happened with the static tests, it is important to know the memory footprint of any of these interpreters. **Table A.9** shows the memory which is being used by some of the programs that can execute PHP.

Table A.9: Web server PHP memory footprint

	VmRSS (kB)	VmSize (kB)
Apache mod_php	5012	241720
FastCGI	3748	170908
PHP-FPM	4140	221372

As said previously, PHP can be executed by means of different interpreters, such as kernel loaded module, CGI, FastCGI, and PHP-FPM. Kernel module it is only available in one of the tested configurations, which is Apache Prefork. In this case, it is loaded a PHP interpreter for any new process that is being created. Memory footprint of Apche's mod_php module is high. It has a total of 5MB of allocated memory in RAM and up to 241MB of virtual memory assigned. This process yet includes the Apache's process, as it has an interpreter loaded into each Apache's processes. Nevertheless, it is a high memory footprint, limiting the number of maximum processes to spawn.

FastCGI processes are spawned apart, meaning that those processes need to be started manually in order to generate the PHP interpreter. It is an evolution of CGI, where it is not being killed each time a request is answered. Memory footprint is lower than mod_php, with a total allocate RAM memory of 3.7MB and 171MB of virtual memory. It is possible to control how many FastCGI processes to spawn when launching the process, so the user can control the memory which is being consumed.

PHP-FPM is an evolution of FastCGI, which stands for PHP FastCGI Process Manager. It is a process manager that controls each of the PHP FastCGI being created. It has different user accessible directives to customize the execution environment, but also has a larger memory footprint. The total allocated RAM memory is 4.1MB and the assigned virtual memory is 221MB, larger than for FastCGI. Neither FastCGI nor PHP-FPM counts the memory footprint of the web servers, so it is needed to add the memory consumption of each web server.

Back in **section 4.2.1** of this report, it was presented some results about PHP usage. It was possible to see that using a dynamic language introduced a performance penalty, which regarding the web server it was more or less considerable. Apache's mod_php was the best performing web server, with and without keep-alive feature. Also, knowing their memory footprint it is possible to say that Apache is the web server that consumes more memory.

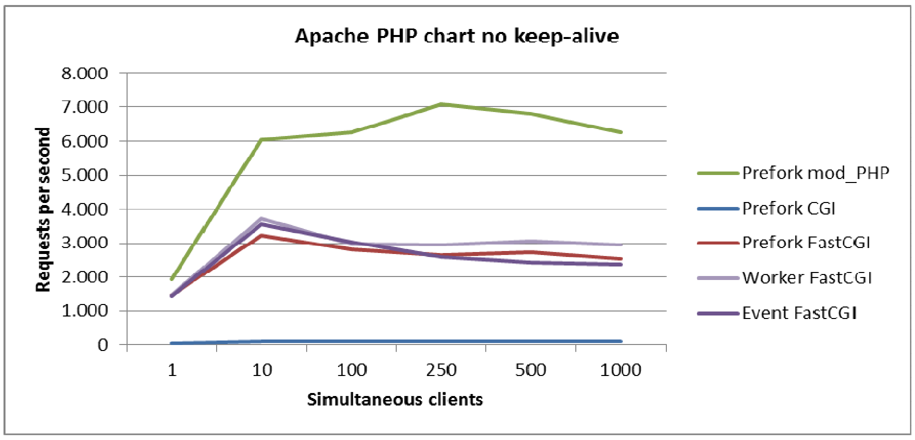

Fig. A.9: Apache PHP no keep-alive chart

Fig. A.9 shows the performance of Apache web server running PHP without keep-alive. It is possible to see that there are three groups. The first one is mod_php, which gets up to 7,000 requests per second. The second group is FastCGI interpreter, reaching up to 2,600 requests per second. Finally, there is CGI mode, which gets less than 100 requests per second. Enabling keep-alive (**section 4.2.1**) does not give any special increase, except for mod_php. In this case, performance increases up to 10,000 requests per second, being very stable across the entire test. FastCGI and CGI performance got the same result than before, meaning that in this case keep-alive does not affect FastCGI or CGI performance. The best way of running PHP in Apache is by using mod_php, but it is important to check the memory footprint of the web server. Eventually, it will be a trade-off between performance and memory footprint.

Event-driven web server became interesting in previous tests, where using different active worker configurations generated more performance. Using PHP by means of an external program, affects the overall performance, so it is not expected to see any high increase of performance between configurations.

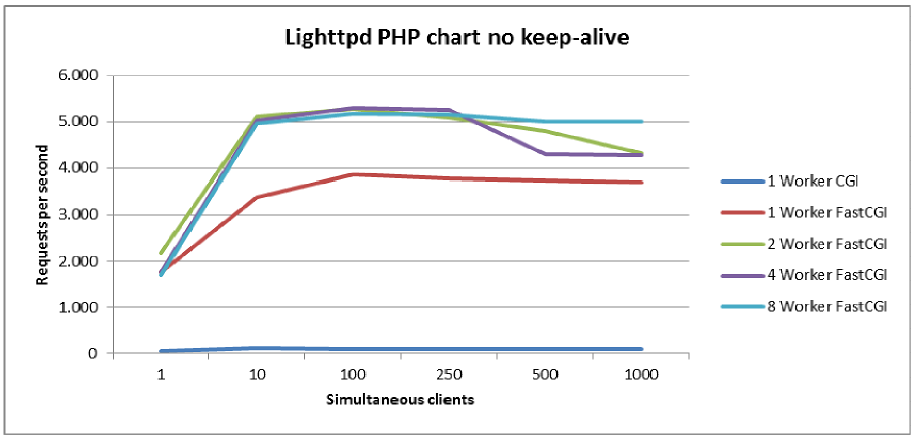

Fig. A.10: Lighttpd PHP no keep-alive chart

Fig. A.10 shows the results of Lighttpd PHP no keep-alive chart. It is possible to see three different groups. At the top of the chart are the configurations with more than one Lighttpd process, taking advantage of the multi-process capabilities of the system. The top performance gets up to 5,000 requests per second, but is not as stable as the number of simultaneous clients increase. Although it is possible to see that 8 worker configuration has fewer requests per second, but is it more stable. Afterwards, it is Lighttpd configuration with only one worker. Here, top performance is less than 4,000 requests per second, although it is very stable across the entire test. It is possible to see a big advantage of using more than one worker at the same time. Finally, using CGI protocol it only reaches 100 requests per second, which is the slower mark of the test.

In **section 4.2.1** was presented the Lighttpd PHP keep-alive chart that was useful to see the performance differences by using multi process approach. Performance does increase a little when enabling keep-alive feature, up to 1,000 requests per second in the best case, although performance is less stable when increasing the number of simultaneous clients.

Nginx is the other event-driven web server architecture. Performance expectance is like Lighttpd, although, as it is shown in **Fig. A.11**, it becomes less stable as increasing the number simultaneous clients. With a two worker configuration, Nginx performance is not the best at the beginning of the test, but it remains stable during the rest of the test, even though when reaching 1,000 simultaneous clients. Performance is about 4,500 requests per second. One worker configuration is not the best performing configuration for Nginx, although it is stable during the entire test, but it gets less than 4,000 requests per second. CGI protocol is not supported by Nginx, so it cannot be tested. Regardless not testing CGI protocol, the expected performance would be the same as in previous cases.

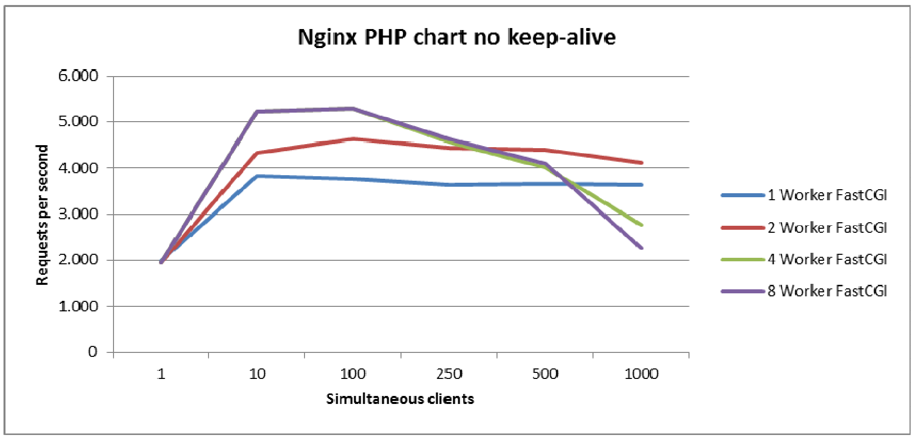

Fig. A.11: Nginx PHP no keep-alive chart

At the beginning of the PHP test case results (**section 4.2.1**), it was possible to see that Cherokee web server had an issue as simultaneous clients increase above 100. In order to solve this performance issue, it was done some changes in its configuration. Running PHP can be done by using FastCGI and also with PHP-FPM (PHP FastCGI Process Manager). It was tested varying the number of threads that Cherokee can execute, the interpreter, as well as changing the socket type. Cherokee is a single process multithread web server, so limiting the number of threads will limit the web server's overall performance. By default, Cherokee creates 10 threads to service the incoming requests. Sockets also play an important role in FastCGI support, as they connect the web server with the PHP interpreter. There are different socket types, UNIX and TCP. UNIX socket can deliver better performance than TCP, because they do not need the extra overhead of going through the TCP/IP stack.

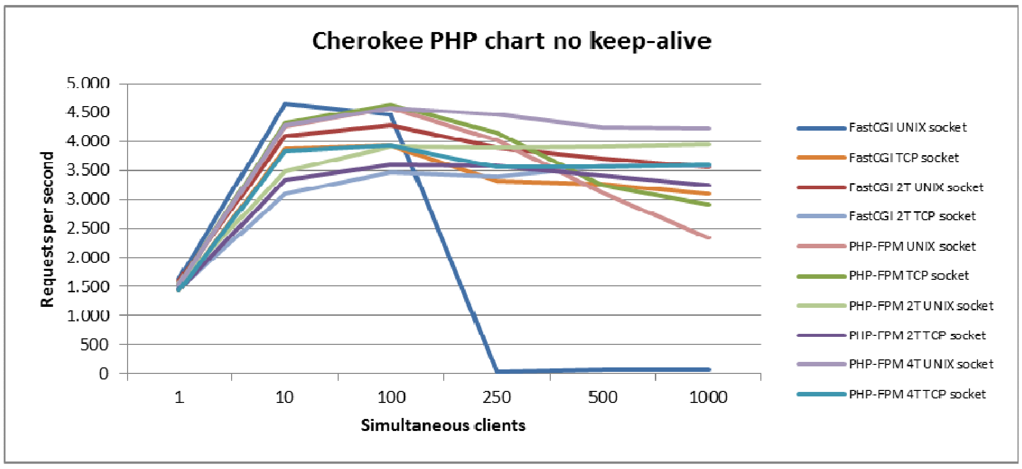

Fig. A.12: Cherokee PHP no keep-alive chart

This test consisted in five different configurations regarding the use of UNIX or TCP socket. **Fig. A.12** shows the obtained results. It is important to notice the performance difference when using UNIX or TCP sockets paired with FastCGI. In UNIX case it is possible to see how it loses all its performance, due to Cherokee getting the 200% of the web server processor (remember it is a dual core machine); meanwhile, in TCP mode it is more or less stable between 4,000 and 3,500 requests per second.

The best performing case is PHP-FPM using four threads and UNIX sockets. It is possible to see that top performing modes are UNIX based sockets, using PHP-FPM. PHP-FPM is way more configurable than php-cgi running in FastCGI mode, letting the user a more custom execution environment. Furthermore, TCP sockets have lower performance compared to UNIX ones, but it will probably perform better under stressed systems.

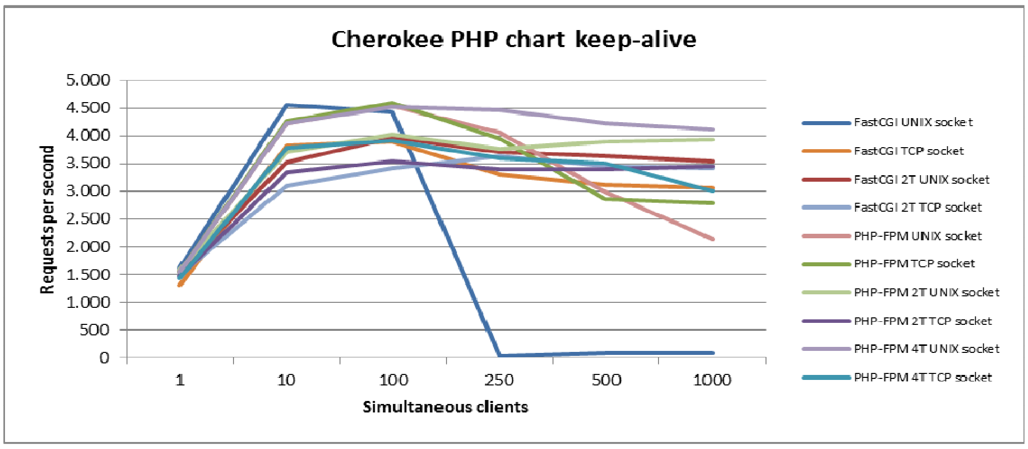

Fig. A.13: Cherokee PHP keep-alive chart

Using keep-alive does not have a great impact in performance results of the test; indeed, it has almost the same behavior. As it is shown in **Fig. A.13**, default configuration of Cherokee using FastCGI and UNIX socket gets the same performance issue. Limiting the number of threads, as well as using PHP-FPM manager to load PHP web pages provokes a more stable performance results.

A.3.2.2 Python

Python, which is another dynamic language, is being used by different frameworks to provide dynamic web experience. It also has a lot of different connectors to other programs of frameworks, like in PHP. Executing Python can be done in different ways, but always using and external interpreter, like FastCGI, CGI or uWSGI. The three of them are explained in detail in **Chapter 2**.

As happened in previous sections, it is important to know the memory footprint of each one of the execution environments. **Table A.10** shows the different memory consumptions for each one of the tested Python interpreters.

Table A.10: Web server Python memory footprint

	Worker		Master	
	VmRSS (kB)	**VmSize (kB)**	**VmRSS (kB)**	**VmSize (kB)**
Apache Prefork (mod_uWSGI)	5088	377116	10224	377116
Apache Worker (mod_uWSGI)	2196	382936	35052	104244
Apache Event (mod_uWSGI)	2188	366420	18524	87728
Lighttpd FastCGI	808	52472	5412	179264
Nginx uWSGI	7196	47240	896	41084
Cherokee FastCGI	4852	142792	752	25868
Yaws FastCGI	41400	171464	-	-
uWSGI	2588	81708	-	-

Apache's configurations support the three different modes of execution, CGI, FastCGI and uWSGI. Although, it is only tested with uWSGI. Memory footprint of Apache Prefork configuration is another time high, with a total of 5MB and 377MB of maximum virtual memory. Worker and Event configurations have very similar behavior, as they share some of the environment. Master process of the three configurations consume more RAM memory, and have less virtual memory assigned, except for Prefork mode which is almost the same as before.

Lighttpd and Nginx web servers do have almost the same memory footprint, as they do not run any new kernel module to run uWSGI or FastCGI. The same happens to Cherokee and Yaws. The four web servers only load more instructions in order to pass information to an external program, by means of a socket. uWSGI process is spawned manually by the user, with a memory consumption of 2.5MB and maximum virtual memory of 82MB per process.

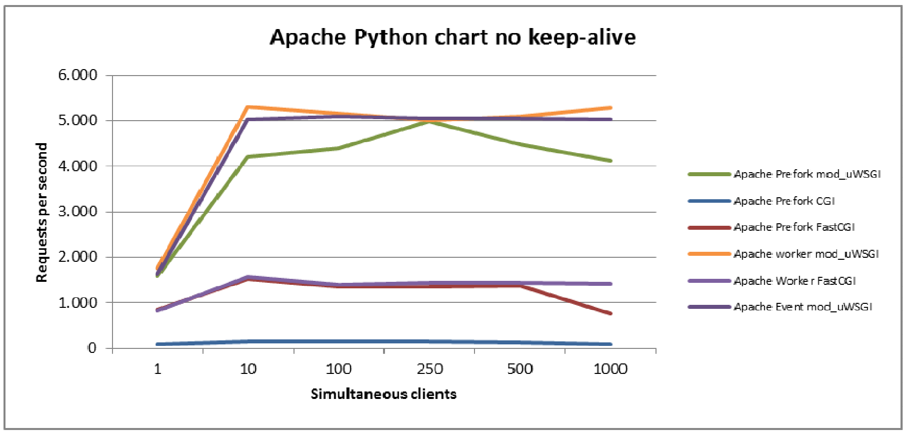

Fig. A.14: Apache Python no keep-alive chart

There is a clear performance difference between the three used interpreters in Apache. As shown in **Fig. A.14** the top performing are those using uWSGI, getting up to 5,000 requests per second. Secondly, there is FastCGI protocol. Meanwhile, in third place is CGI protocol with less than 100 requests per second. uWSGI is a very good option to run Python code, as it has far better performance and also a customizable execution environment. Taking into account results obtained in **section 4.2.2**, not using keep-alive does not produce any noticeable performance penalty.

Nginx's performance is also limited by the uWSGI interpreter (see **Fig A.15**). It is possible to see that, although there is the possibility to use the multi process configuration, it does not report any special performance increase. Performance varies as simultaneous clients increase, with a downward trend. Regarding the results obtained back in **section 4.2.2**, not enabling keep-alive produces slightly more variation, although both have the same performance.

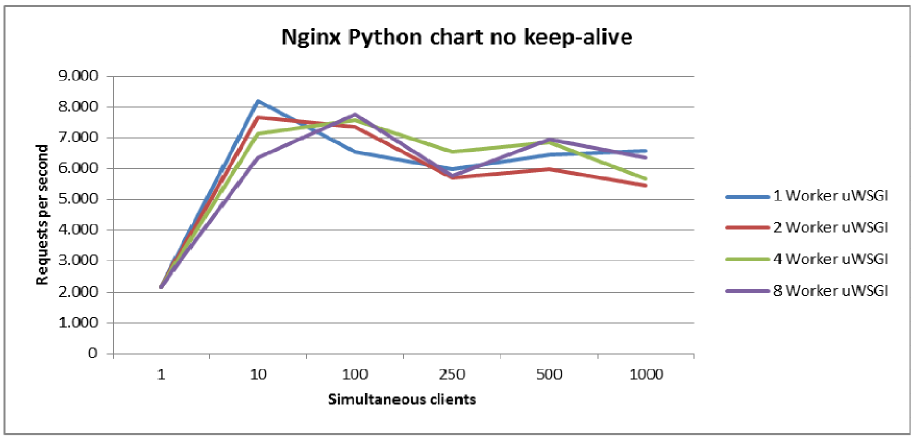

Fig. A.15: Nginx Python no keep-alive chart

Cherokee is the last of the three web servers that uses uWSGI interpreter to run Python. Moreover, it has an estrange behavior when increasing the number of simultaneous clients above 100 users. This performance decrease is very similar to what is found during PHP test case. In this case, configuration variations are only done with UNIX sockets, as TCP sockets introduce a high performance penalty.

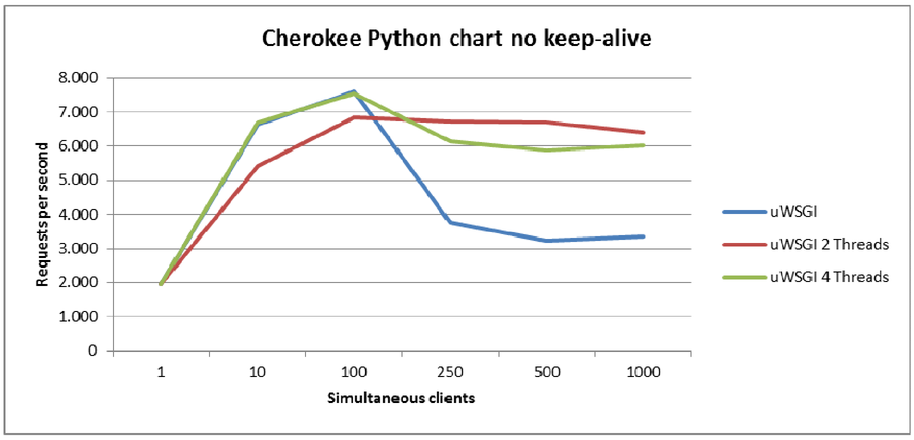

Fig.A.16: Cherokee Python no keep-alive chart

Fig. A.16 compares the different performance results of limiting Cherokee's number of threads. It is possible to see a big improvement by limiting the number of threads to two or even four. Using more threads increases the performance when there are less than 100 clients in the system; meanwhile, using fewer threads keeps stable the performance when there are more than 100 simultaneous clients. It is important to highlight that this effect is produced when requesting a small file, which eventually the bottleneck will be the web server itself, not the interpreter. With larger frameworks, with hundreds of lines of codes and functions, those issues would not repeat, as the interpreter would be the bottleneck.

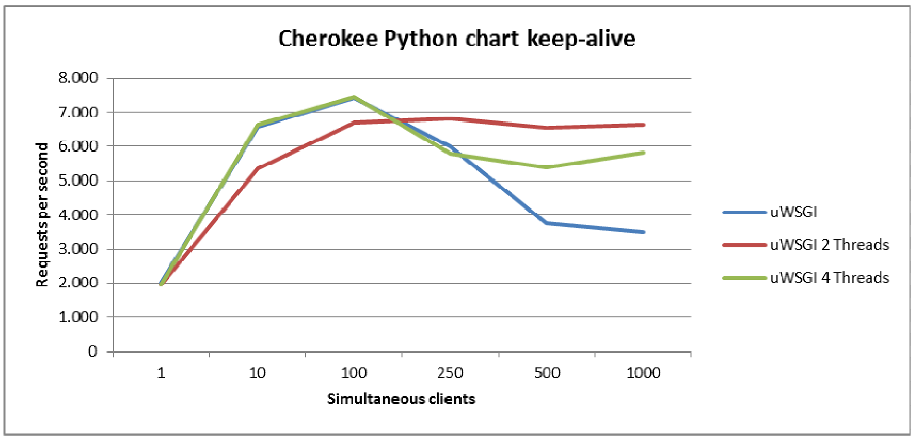

Fig. A.17: Cherokee Python keep-alive chart

Enabling keep-alive feature, gets almost the same performance in the three cases, see **Fig. A.17**. Default uWSGI configuration decreases its performance although, not as quick as before, but being noticeable anyway. As said before, this issue is produced when the web server is the bottleneck, when it has to process lots of requests in a short period of time.

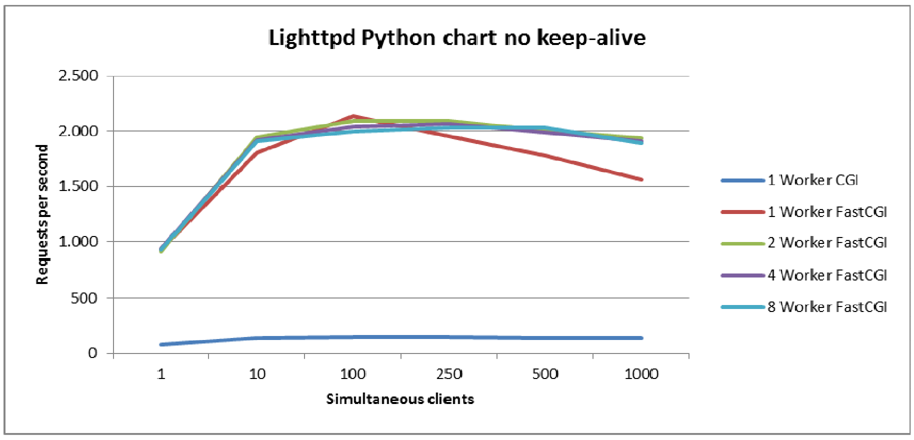

Fig. A.18: Lighttpd Python no keep-alive chart

Lighttpd's performance in different configurations is almost the same among them. More than one worker configuration gets better performance as simultaneous clients increase in the system, above 250 users. There is not so much difference in terms of requests per second, as in average it gets 1,700 requests per second. If compared with keep-alive chart (**section 4.2.2**) it is possible to see that there are not noticeable differences.

A.3.2.3 Servlet and JSP

Servlet and JSP dynamic web protocols are part of Java technologies, which are found in only one of the tested web servers, Tomcat. Java programming language is very used in the Internet, as well as in desktop software to provide very rich applications with high portability. Servlet and JSP test cases are dynamic web protocols; although, the obtained performance is as high as HTML tests.

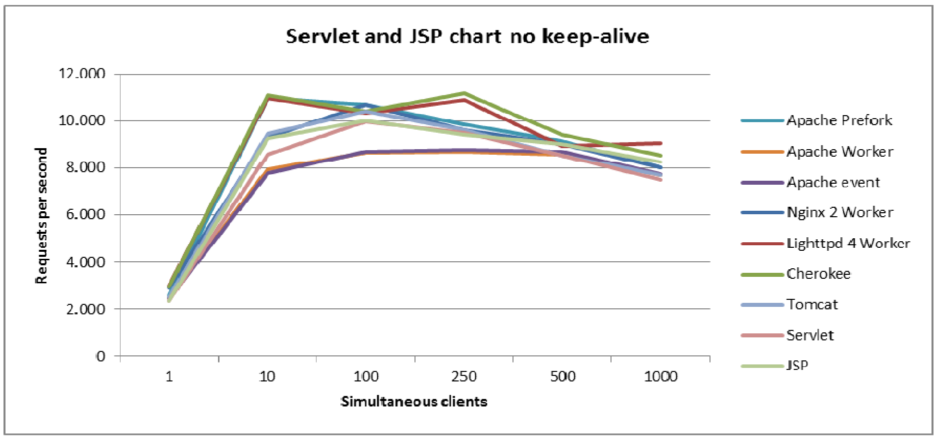

Fig. A.19: Servlet and JSP vs HTML chart no keep-alive

Fig. A.19 shows Servlet and JSP results compared with HTML values. Without using keep-alive, the obtained performance is higher than for some Apache's configurations, and near of Nginx and Tomcat. It delivers up to 10,000 requests per second for both, Servlet and JSP. Although, performance is not stable when increasing the number of simultaneous clients in the system. It is possible to compare with other HTML web servers as they get almost the same performance.

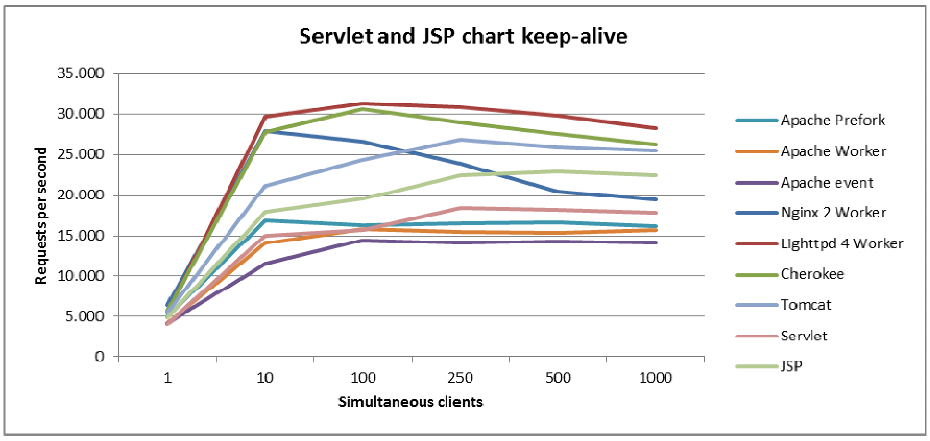

Fig. A.20: Servlet and JSP chart keep-alive

Enabling keep-alive, increases the performance for all the web servers. As shown in **Fig. A.20**, performance increased for both Servlet and JSP, up to 18,400 and 22,400 requests per second, respectively. Both cases overtake the performance obtained by Apache web server, but fall behind of other servers such as Cherokee, Nginx or Lighttpd.

Nevertheless, Servlet and JSP power dynamic web pages meanwhile, HTML only static pages. So, they cannot be compared, as Java technologies offer desktop class features to web pages. Eventually, this comparison must be done with PHP and Python using CGI and FastCGI. **Fig. A.21** shows the relative performance between Servlet, JSP and PHP. It is possible to see the difference in performance for Servlet and JSP in front of the other cases. The best performing PHP web server is Apache's mod_php with as much as 7,000 requests per second, falling behind of Servlet and JSP, which rates up to 10,000 requests per second. In any case, performance is far superior to FastCGI or CGI.

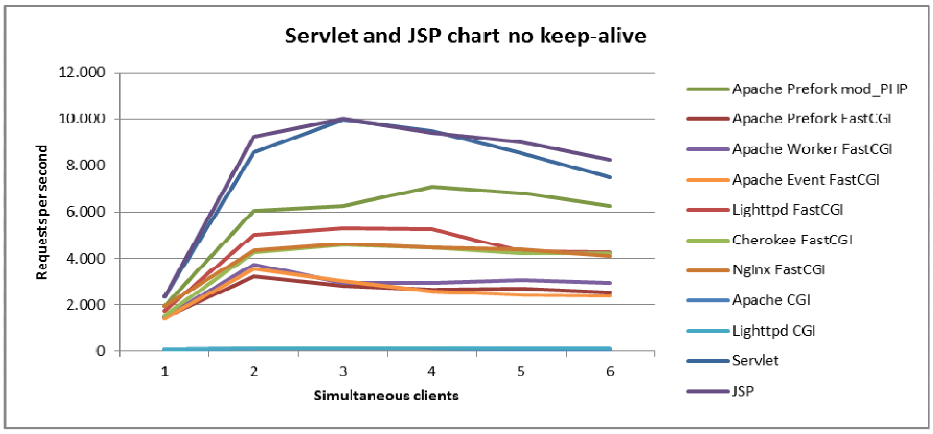

Fig. A.21: Servlet and JSP vs PHP chart no keep-alive

Moreover, enabling keep-alive, performance increases even more for Servlet and JSP, getting up to 18,400 and 22,400 requests per second, respectively (see **Fig. A.22**). Mod_php is another time the best performing PHP web server, with up to 10,000 requests per second, although being slower than Servlet and JSP. The advantage of Servlet and JSP is that for both cases

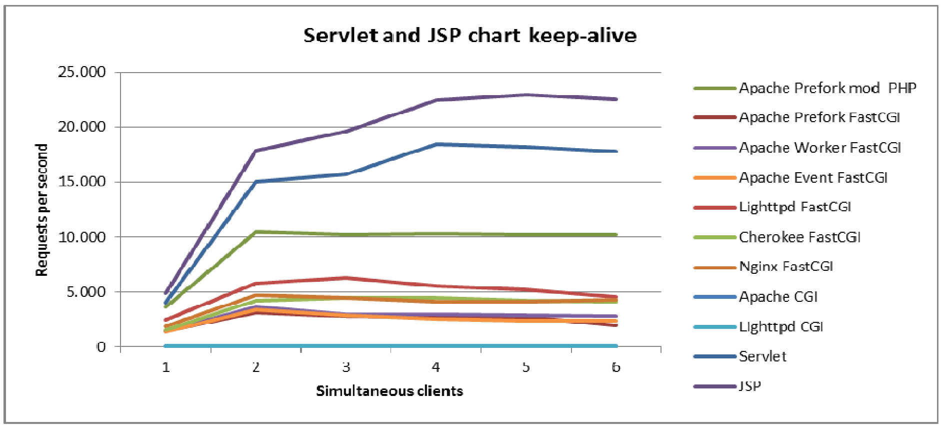

Fig. A.22: Servlet and JSP vs PHP chart keep-alive

A.3.3 HTTPS

Secure HTTP is a very important feature in nowadays communications. In this case, it is being tested almost all of the web servers in HTTPS mode, using SSL and TLS protocols to communicate securely between the two hosts.

As in previous tests, memory footprint is important as it will show the memory consumption of each of the tested web servers. Overall, the memory footprint for each web server of this report is slightly higher when configured with SSL/TLS support. This higher memory consumption is only of few KB or even few MB.

Table A.11: Web server SSL memory footprint

	Worker		Master	
	VmRSS (kB)	VmSize (kB)	VmRSS (kB)	VmSize (kB)
Apache Prefork	3284	76420	3284	76420
Apache Worker	2572	387880	36192	109188
Apache Event	2572	371364	19664	92672
Lighttpd	796	53892	796	53892
Nginx	7276	57616	964	51460
Cherokee	-	-	-	-
Yaws	41504	171976	-	-

A.3.3.1 HTTPS HTML

If compared to the first HTML results, enabling SSL results in a considerable performance penalty (see **section 4.3**). Regarding the architecture that web servers are being using, this decrease is more notable. In Apache's case, without keep-alive there is an important performance penalty of using SSL, see **Fig. A.23**. In this case, it does not matter which configuration is used, as all of them have a similar performance.

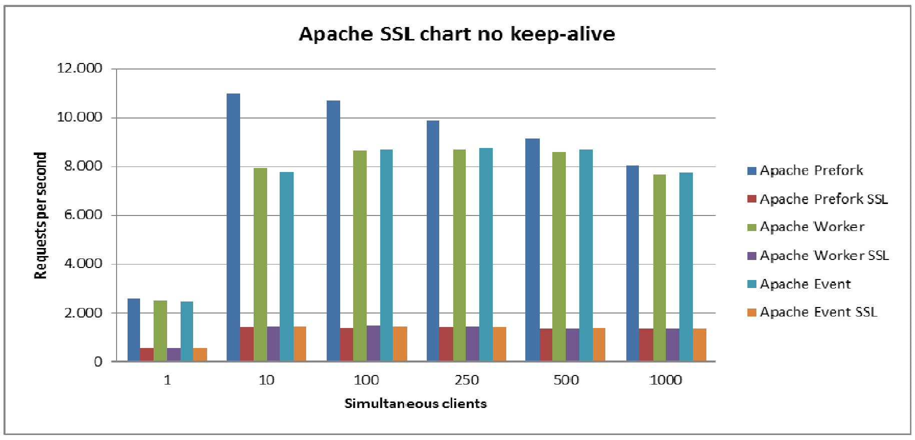

Fig. A.23: Apache HTTPS no keep-alive comparison chart

Apache's performance enabling keep-alive increases in both cases, with and without SSL support. It is possible to see in **Fig. A.24** that the gap between both scenarios is reduced, having a lower performance penalty when enabling SSL.

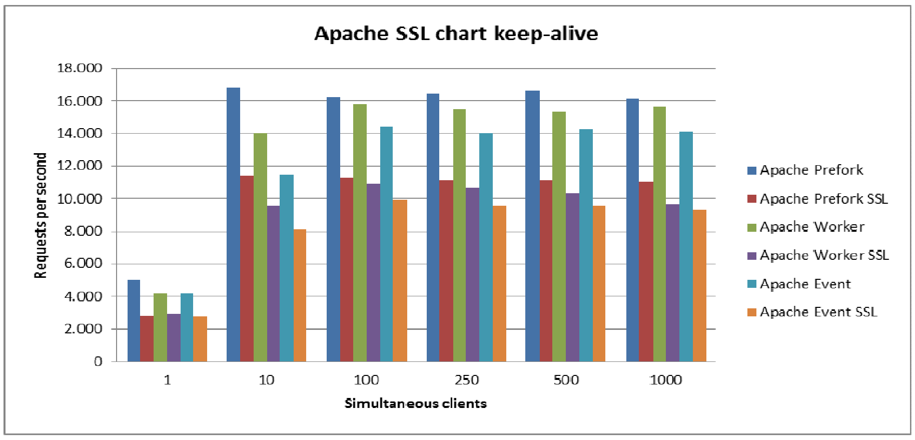

Fig. A.24: Apache HTTPS keep-alive comparison chart

As it is shown in **Fig. A.25**, disabling keep-alive in Lighttpd and using SSL, has a great impact in the overall performance of the web server, going down from about 10,000 requests per second (in average) to 1,500 requests per second.

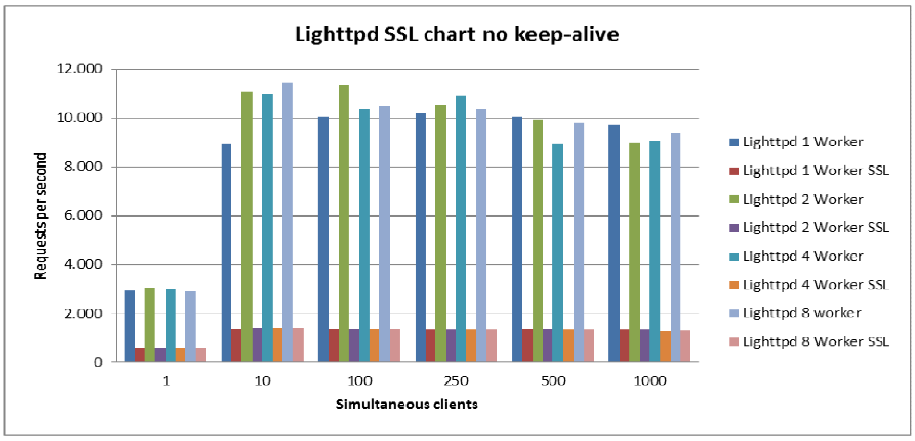

Fig. A.25: Lighttpd HTTPS no keep-alive comparison chart

Enabling keep-alive feature in Lighttpd provokes an increase of performance in both cases, with and without SSL enabled (see **Fig. A.26**). Performance of SSL HTML does not have a very high variation along the entire test, but it is possible to see the performance penalty. This penalty is even higher when Lighttpd is configured with more than one active worker, where the gap between both test cases is bigger.

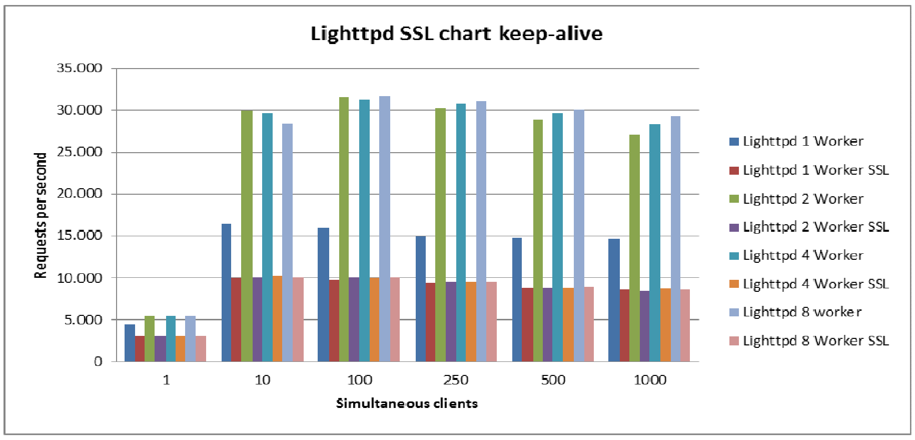

Fig. A.26: Lighttpd HTTPS keep-alive comparison chart

Finally, Nginx has a very similar performance than Lighttpd's case, with some variations of the performance of HTML results regarding the number of active workers (see **Fig. A.27**). SSL performance results are very similar of those gotten from Lighttpd, and performance penalty by enabling SSL connections is very noticeable.

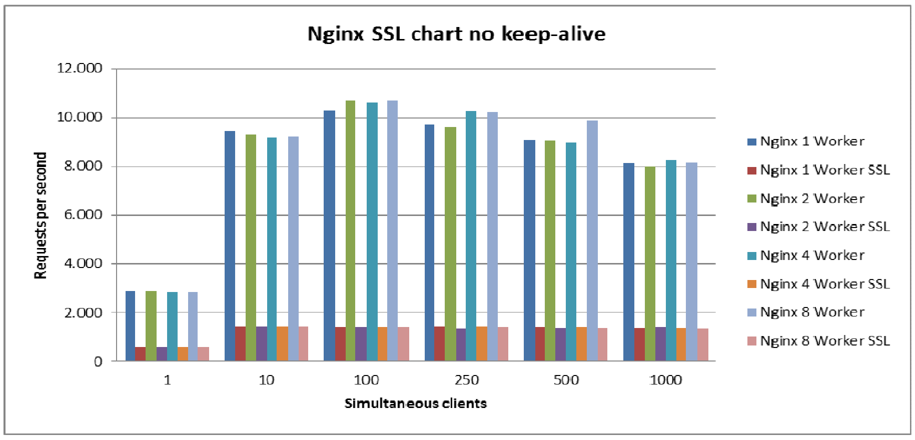

Fig. A.27: Nginx HTTPS no keep-alive comparison chart

Meanwhile, enabling keep-alive (**see Fig. A.28**) increases the performance results of both scenarios (with and without SSL), reaching up to 15,000 requests per second. Although, it is not stable across the entire test, it reduces the penalty gap between both test cases. Nginx's performance is even higher than Apache's for SSL.

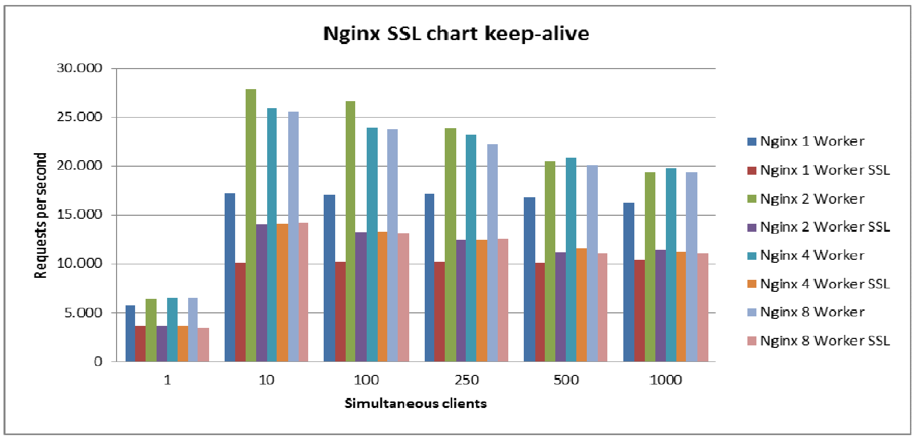

Fig. A.28: Nginx HTTPS keep-alive comparison chart

A.3.3.2 HTTPS PHP

Next charts give the possibility to show relative performance results between each web server using SSL and PHP. Apache's HTTPS comparison chart (see **Fig. A.29**) without keep-alive shows that SSL feature limits the performance of the web server, around 1,400 requests per second, meanwhile with Prefork using mod_php without SSL the performance is around 6,000 or even 7,000 requests per second. Apache's Worker and Event configurations do not get such a high performance, and the decrease by enabling SSL is around 150%.

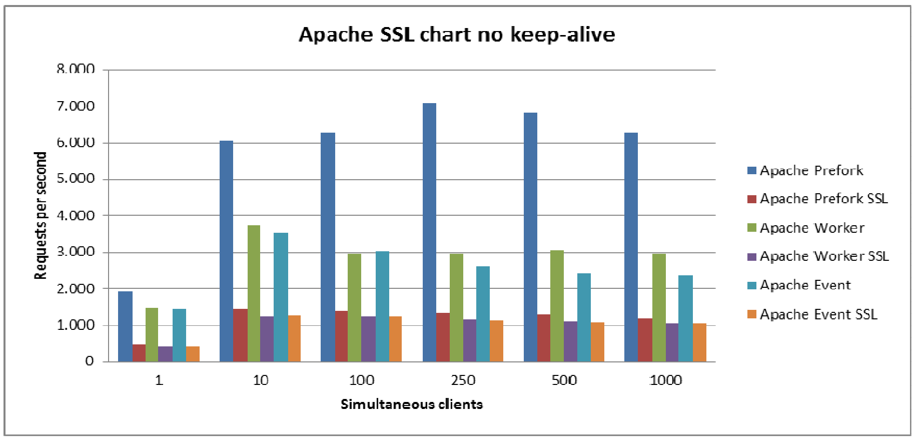

Fig. A.29: Apache HTTPS PHP no keep-alive comparison chart

Keep-alive does not introduce any performance increase. It is possible to see that using such feature mod_php without SSL increase its performance; although, it does not increase when running secure communications. Performance penalty for mod_php configuration is around 600%, meanwhile for the other two configurations is 150%.

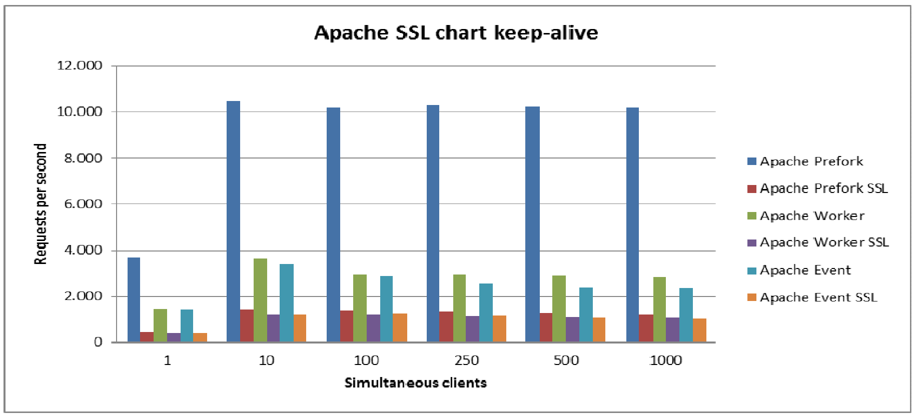

Fig. A.30: Apache HTTPS PHP keep-alive comparison chart

Lighttpd's case is different than for Apache. Without keep-alive, performance results get the same rate as in the other SSL tests, around 1,400 requests per second (see **Fig A.31**); but, when enabling keep-alive feature performance increases. This boosts of performance gets up to 4,000 requests per second, which is somewhat surprising (see **Fig. A.32**).

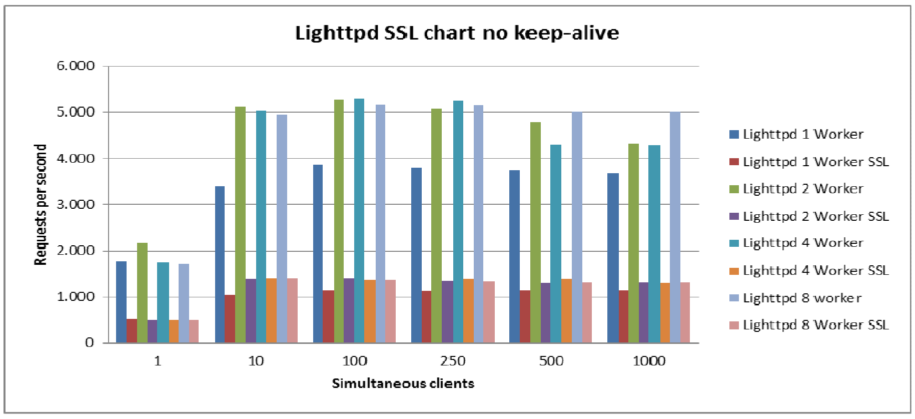

Fig. A.31: Lighttpd HTTPS PHP no keep-alive comparison chart

Increasing the number of active workers does have a positive impact on the overall web server performance, which is an important feature.

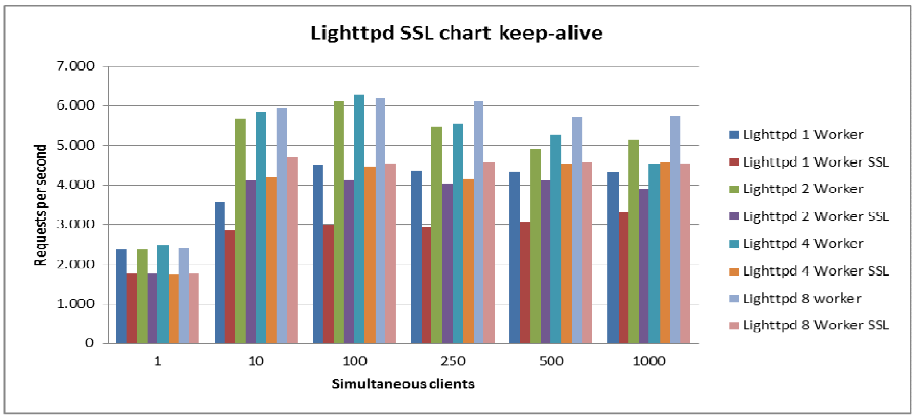

Fig. A.32: Lighttpd HTTPS PHP keep-alive comparison chart

The other web server to analyze is Nginx. Performance obtained without using keep-alive is as expected; around 1,400 requests per second, which is maintained across the entire test (see **Fig. A.33**). Instead of gaining some performance when enabling keep-alive, it maintains the same as before, as it is possible to see in **Fig. A.34**, like in Apache's case.

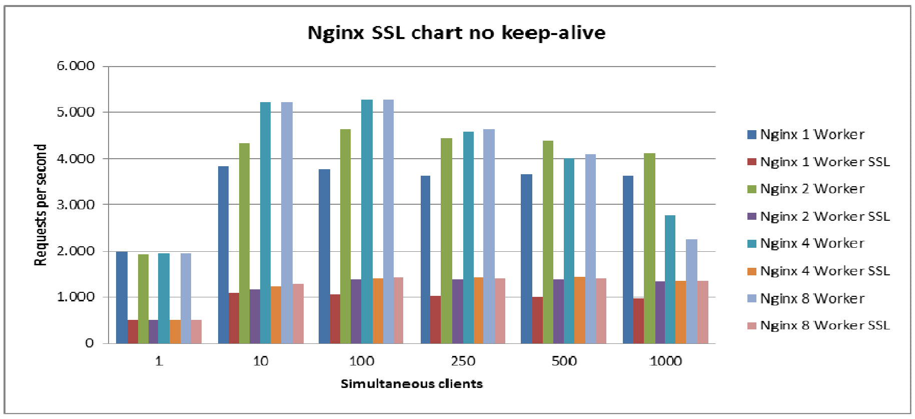

Fig. A.33: Nginx HTTPS PHP no keep-alive comparison chart

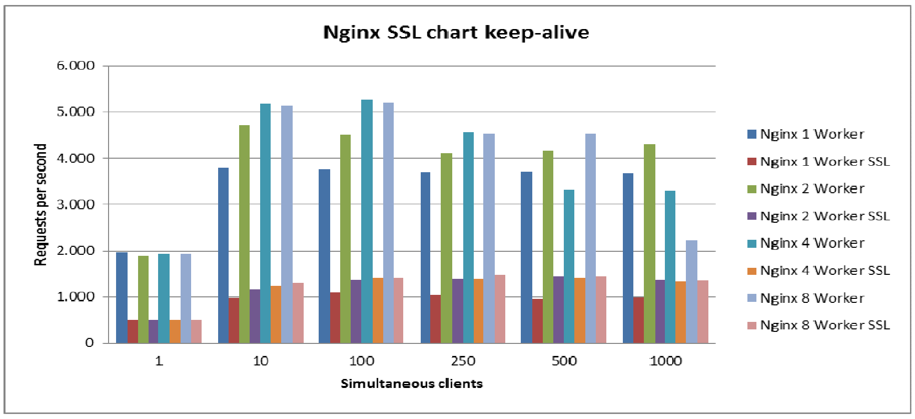

Fig. A.34: Nginx HTTPS PHP keep-alive comparison chart

A.3.3.3 HTTPS PYTHON

Python is another tested dynamic language. In order to compare the obtained performance when using SSL against not using it, it is useful to present different charts about each web server. In those charts it will be possible to compare different server's configurations in different situations.

Apache is the first tested web server. In this case, and as before, performance result by enabling HTTPS is limited to 1,400 requests per second (see **Fig. A.35**). It is possible to see a high performance penalty, around 72%.

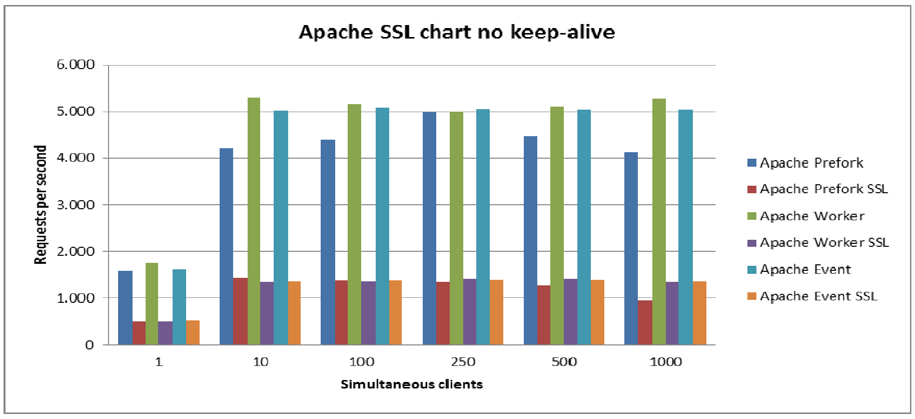

Fig. A.35: Apache HTTPS Python no keep-alive comparison chart

Enabling keep-alive to Apache's test does not represent any performance improvement, as the obtained results are almost the same as before, as it is shown in **Fig. A.36**. Performance penalty is as before, about 72%.

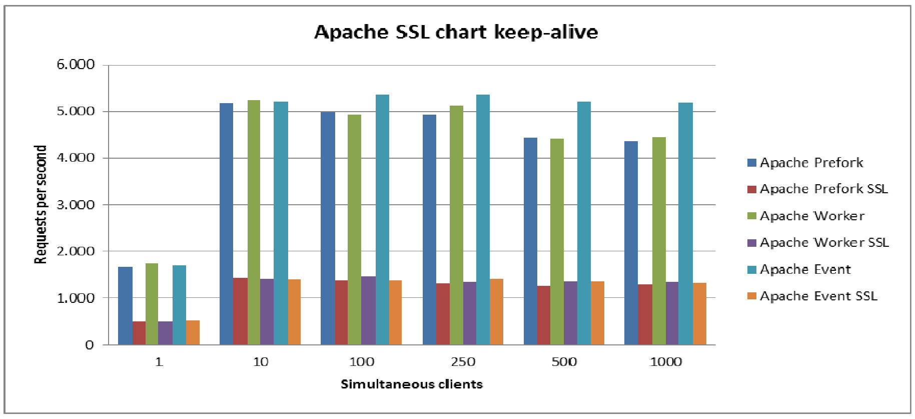

Fig. A.36: Apache HTTPS Python keep-alive comparison chart

In Lighttpd's web server, performance associated to HTTPS test case is the same as before, with only 1,400 requests per second (see **Fig. A.37**). Compared with not using SSL, performance penalty is 35%, less noticeable than for Apache's.

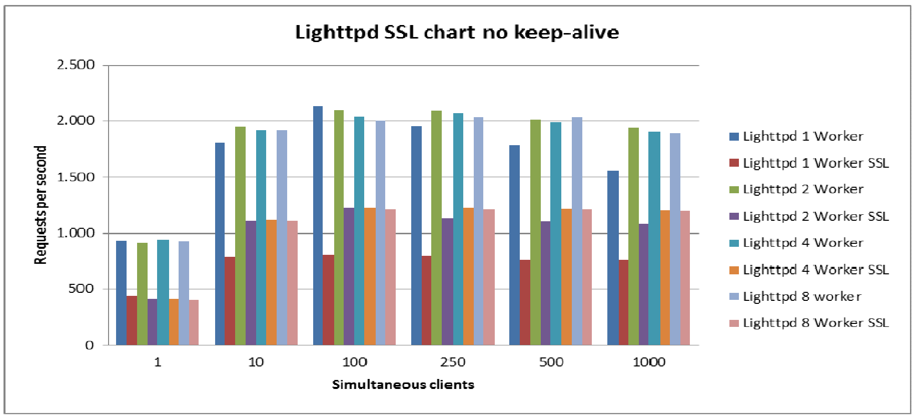

Fig. A.37: Lighttpd HTTPS Python no keep-alive comparison chart

Using keep-alive with Lighttpd increases its performance, up to 1,800 requests per second, which is an increase of a 38% (see **Fig. A.38**). Compared with the no SSL test case, it is possible to see that the performance is very similar, without any noticeable penalty.

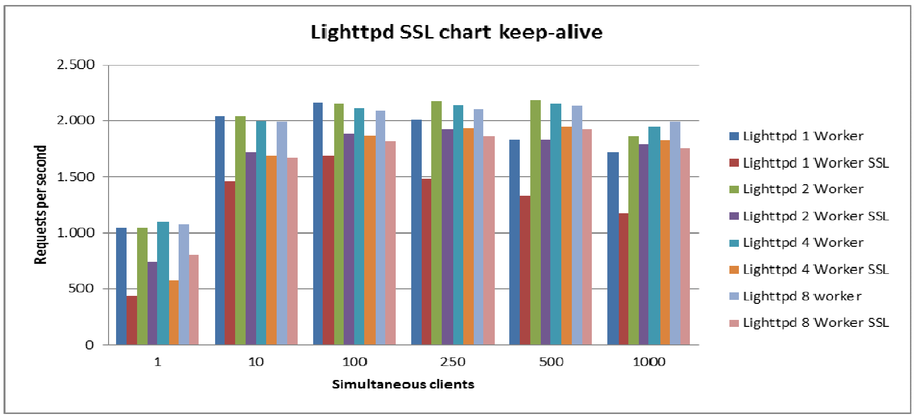

Fig. A.38: Lighttpd HTTPS Python keep-alive comparison chart

Nginx is the last event-driven web server. Performance enabling SSL is the same as the previous web servers, 1,400 requests per second (see **Fig. A.39**). Although, it is possible to see a high performance penalty by enabling secure navigation, around 78%, which is very noticeable.

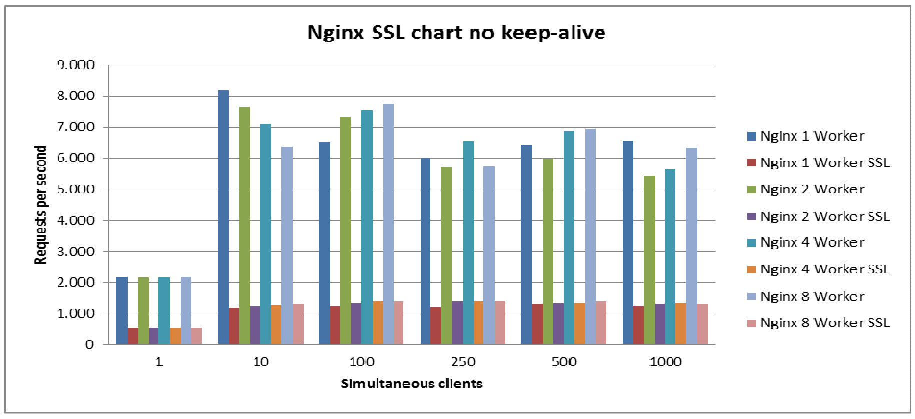

Fig. A.39: Nginx HTTPS Python no keep-alive comparison chart

Nginx's SSL keep-alive test does not represent any performance improvement, as it maintains the same performance results obtained before, as it is possible to see in **Fig. A.40**. Compared with disabling SSL, penalty introduced is the same as before, around 78%, which is very noticeable.

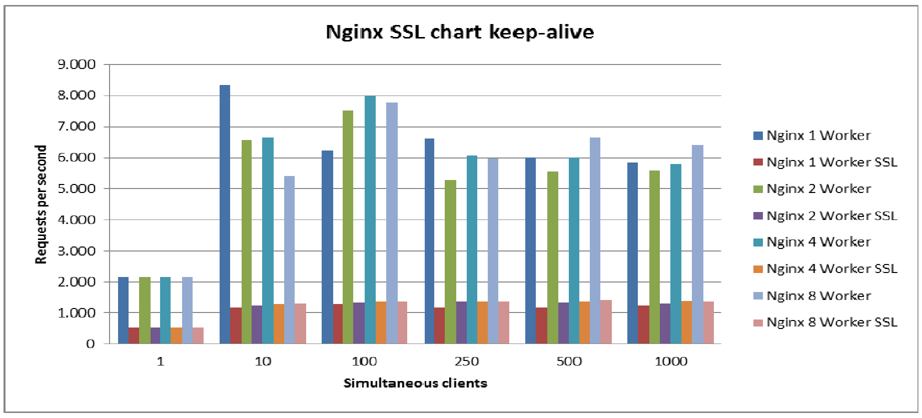

Fig. A.40: Nginx HTTPS Python keep-alive comparison chart

A.3.7 Load test

As said in **Section 4.4**, this test consists in benchmarking the hardware with a Wordpress blog installation. This blog site bases its functionalities in HTML, PHP and MySQL database. In order to present it as a more real performance test, it is installed different programs to increase its overall performance, such as content cache and PHP accelerators. Content cache will be very useful, as the server will cache the requested content and serve it quicker than accessing to the hard disk each time an incoming request arrives.

PHP accelerator will convert any PHP instruction which is requested to op-code, Instead of recompiling the code for every request; it is possible to keep the op-code in memory and save the compile step, gaining some performance.

Table A.12: Memory footprint of PHP accelerator programs

	Before loading main page		**After loading main page**	
	VmRSS (kB)	**VmSize (kB)**	**VmRSS (kB)**	**VmSize (kB)**
mod_php	5012	241720	8308	242748
mod_php with APC	5052	375044	8028	375824
mod_php with Xcache	5072	276700	8904	278352
mod_php with eAccelerator	5020	260268	7884	261040
FastCGI	3748	170908	6684	171824
FastCGI with APC	3792	304232	6328	304888
FastCGI with Xcache	3812	205888	7220	207452
FastCGI with eAccelerator	3764	189456	6176	190112
PHP-FPM	4140	221372	6788	222296
PHP-FPM with APC	4180	354692	6484	355360
PHP-FPM with Xcache	4196	256352	7348	257896
PHP-FPM with eAccelerator	4160	239920	6332	240584

As happened in previous sections of this appendix, it is important to reflect the memory footprint that generated the usage of any web server. In this case it is shown in **Table A.12** the memory footprint of each one of the tested PHP accelerators. Mod_php, FastCGI and PHP-FPM are the standalone releases of PHP interpreters, all of them work with the PHP installation of the server.

Meanwhile, APC, Xcache and eAccelerator, are PHP extensions, which act as a PHP cache. Those extensions must be installed reflecting the PHP installation path as well as adding the extension to the PHP configuration file. Alternative PHP Cache (APC) is supported by PHP group and it will be included in the next release of PHP.

The first interpreter is mod-php. As it is possible to see in Table X.X, its memory consumption is higher than for the others up to 5MB and 8.3MB after loading the main web page. This is because it has the Apache's web server process loaded too. APC accelerator has the highest maximum virtual memory assigned, though it could be a problem in very loaded scenarios. Xcache and eAccelerator have almost the same behavior. As Apache processes increase in memory footprint it lets less free RAM memory to other processes, limiting the overall server's performance.

FastCGI approach could be used by any web server. Being a separated process means also less memory footprint. For this reason, processes only consume 3.7MB and 6 – 7MB once loaded the main blog's page. As FastCGI processes are controlled by the user, who specifies the number of maximum spawned processes, controlling the memory consumption of those is not as critical as before. APC also keeps having the highest virtual memory assigned. PHP-FPM is an evolution of FastCGI, which allows the user to control even more the execution environment of PHP. The memory footprint is higher than for FastCGI (4.1MB and 6.6MB), although lower than for mod_php. Also, the maximum allocated virtual memory is higher.

As seen in **Section 4.4**, performance for this load test is not equal for all of the web servers. Regarding the technology associated to each web server its performance will vary. As for this, Apache Prefork with mod_php does not complete the test, as it runs out of RAM memory as well as CPU. It provokes a huge decrease of performance and starting to timeout requests. Meanwhile, the other web servers accomplish successfully the test, leaving space for further testing.

The first phase of the test, creating of 100 clients per second, does not report any problem for the tested web servers, as all of them support such load. Increasing the number of simultaneous clients, up to 450 clients per second, does reflect some system's limitations, as some of the web servers do get high CPU utilization, as well as high RAM memory consumption. The third phase, which consists in creating 200 simultaneous clients, is performed by all the web servers without any issue.

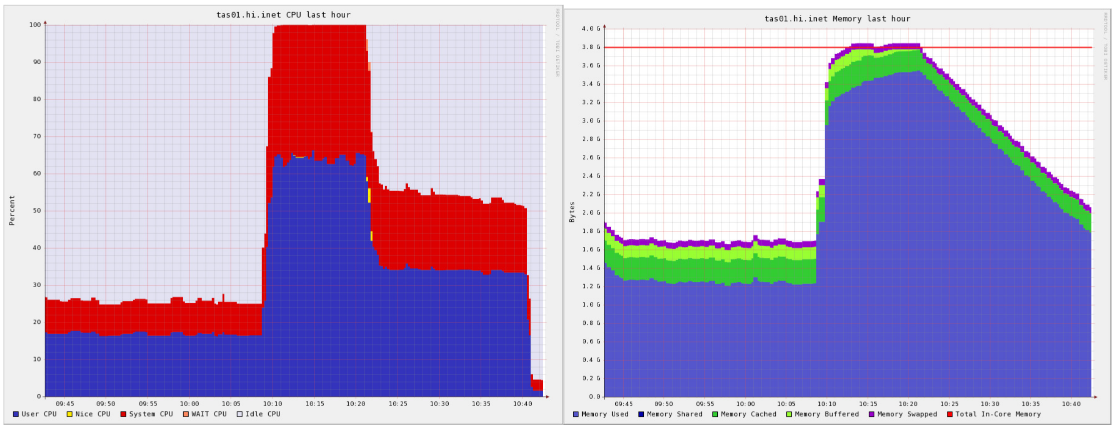

Fig. A.41: Apache Prefork CPU and RAM load

Apache Prefork had some issues during the load test (see **Fig. A.41**). The first phase of the test (100 clients/s) as well as the third phase (200 clients/s) runs without any problem. Nevertheless, during the second phase (generating up to 450 clients per second) the web server reaches the maximum performance of the hardware. CPU load was 100 % and the RAM footprint reaches almost the 4GB available, starting to caching some data into the hard disk. With those two parameters taken into account, there is no space for scale the web server, as there are no hardware resources available. Also, during the second phase of the test, some timeout errors were found, which leads to a not successful test.

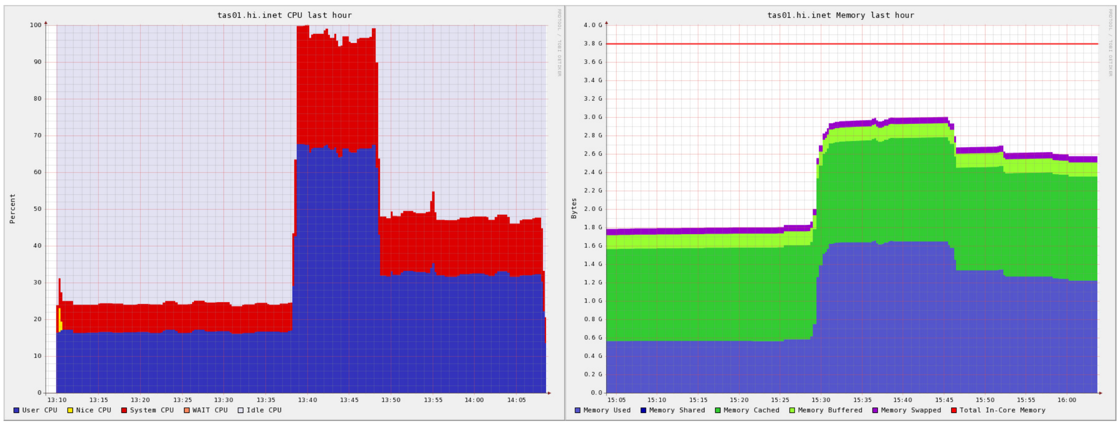

Fig. A.42: Apache Event CPU and RAM load

Apache's Event configuration does work in a very different way than its Prefork sibling. As shown in **Fig. A.42**, CPU load gets up to 100%, although there is still RAM memory available. Compared with Prefork's configuration, Event does have RAM available and does not generate any error during high load phases.

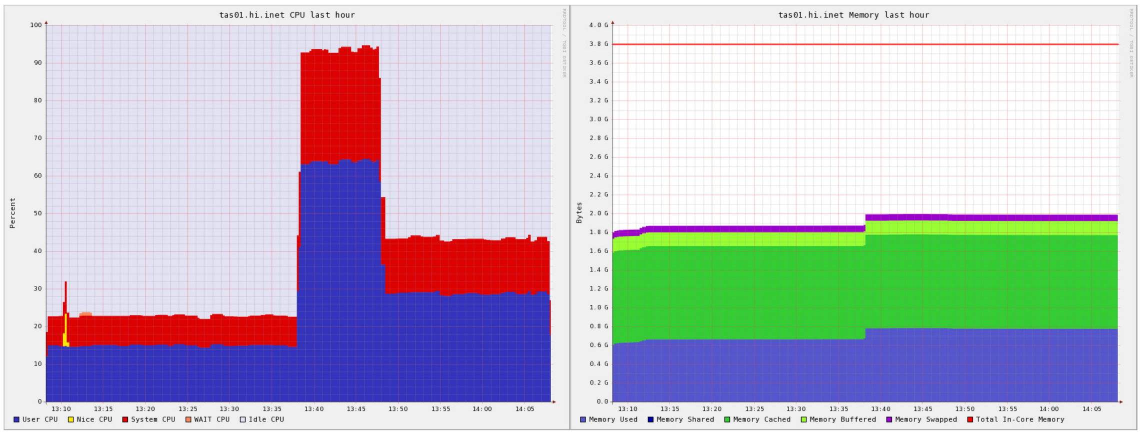

Fig. A.43: Lighttpd CPU and RAM load

Lighttpd CPU load is similar than for Apache's, increasing up to the maximum when the second phase of the tests starts (see **Fig. A.43**). RAM memory footprint during the test is small, only 2GB, being 50% of the available memory in the system. RAM memory is not an issue, so there is some space for scalability, as it is not consuming as much as RAM as the other web servers.

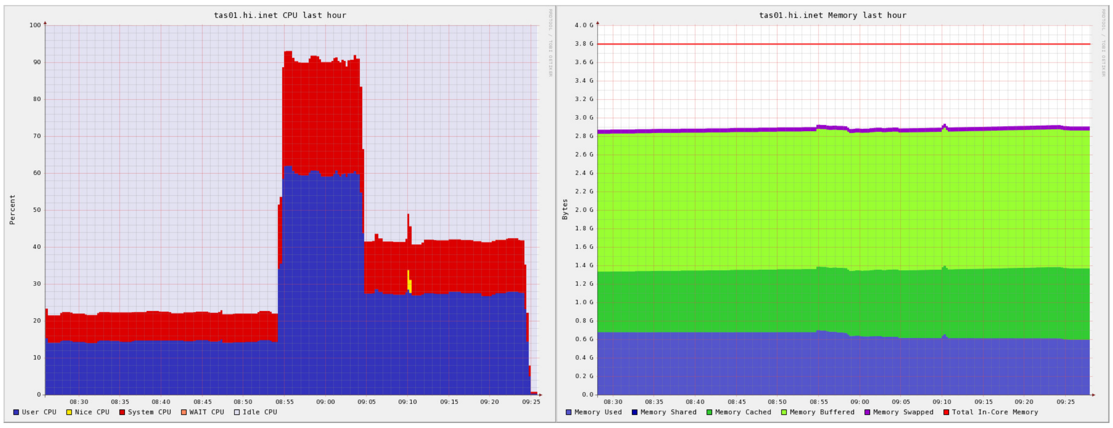

Fig. A.44: Nginx CPU and RAM load

Nginx's hardware performance is very similar to the previous one (see **Fig. A.44**). CPU load is less than 100%, average 90% in the most loaded phase. RAM is almost the entire test with a same value, around 2.8GB. Taking into account the evolution of the CPU load as well as the RAM consumption it is possible to say that it leaves space for further performance increase, with more simultaneous clients.

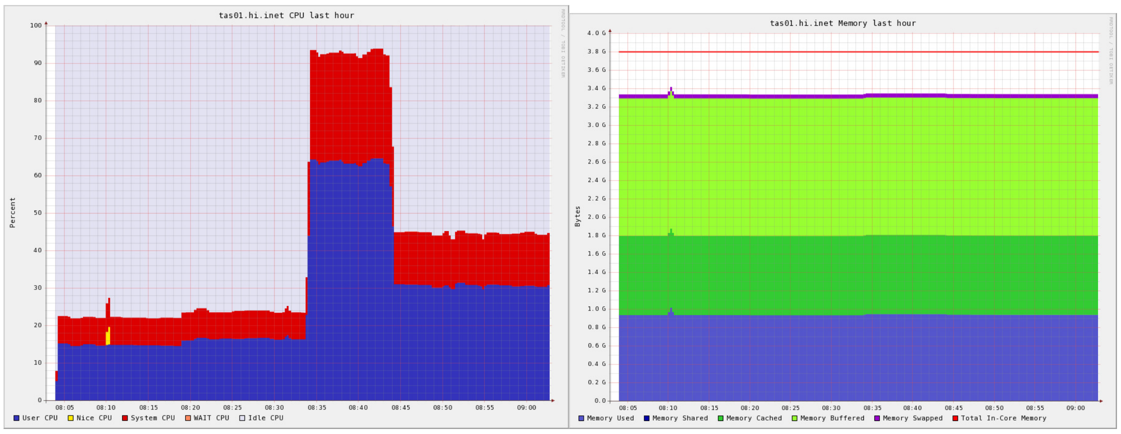

Fig. A.45: Cherokee CPU and RAM load

Cherokee's performance during the load test is also very good. **Fig. A.45** shows its hardware performance, with CPU load reaching up to 92% in the most loaded phase. RAM memory consumption is very high, maintaining stable during the entire test, with 3.4GB. Having such high RAM consumption as well as high CPU load does not leave so much space for scalability, as the system will run out of memory in any time.

As seen during the test, there is a phase in which the load generated does gets the maximum hardware performance. During the second phase, which consisted in creating 450 simultaneous clients, CPU load reaches its maximum point, with up to 100%. Speaking about RAM memory consumption, each web server works different, as some of them maintain the same consumption and other start to consume more. In any case, as simultaneous clients increase, CPU load and RAM consumption increase. Apache Prefork configuration does not finish the test successfully, limited by the unavailability of hardware resources. Meanwhile, other web servers such as Lighttpd, Nginx, Cherokee and Apache's Event do perform it without any issue.